

지네릭스, 열거형, 애너테이션

1. 지네릭스(Generics)

제네릭스 - 클래스 내부에서 사용할 데이터 타입을 외부에서 지정하는 기법

1.1 지네릭스(Generics)란?

- 컴파일 시 타입을 체크해 주는 기능(compile-time type check) – JDK1.5
- 객체의 타입 안정성을 높이고 형변환의 번거로움을 줄여줌
(하나의 컬렉션에는 대부분 한 종류의 객체만 저장)

지네릭스의 장점

1. 타입 안정성을 제공한다.
2. 타입체크와 형변환을 생략할 수 있으므로 코드가 간결해 진다.

1.2 지네릭 클래스의 선언

- 클래스를 작성할 때, Object타입 대신 T와 같은 **타입변수**를 사용 (타입을 변수로 처리)

```
class Box {  
    Object item;  
  
    void setItem(Object item) { this.item = item; }  
    Object getItem() { return item; }  
}
```

```
class Box<T> { // 지네릭 타입 T를 선언  
    T item;  
  
    void setItem(T item) { this.item = item; }  
    T getItem() { return item; }  
}
```

- 참조변수, 생성자에 T대신 실제 타입을 지정하면 형변환 생략 가능

```
Box<String> b = new Box<String>(); // 타입 T 대신, 실제 타입을 지정  
b.setItem(new Object()); // 예러. String이외의 타입은 지정불가  
b.setItem("ABC"); // OK. String타입이므로 가능  
String item = (String) b.getItem(); // 형변환이 필요없음
```

1.3 지네릭스의 용어

Box<T> 지네릭 클래스. ‘T의 Box’ 또는 ‘T Box’라고 읽는다.

T 타입 변수 또는 타입 매개변수.(T는 타입 문자)

Box 원시 타입(raw type)

원시타입
class Box<T> { }
지네릭 클래스

대입된 타입(매개변수화된 타입, parameterized type)

Box<String> b = new Box<String>() ;
지네릭 타입 호출

1.4 지네릭스의 제한

- static멤버에는 타입 변수 T를 사용할 수 없다.

```
class Box<T> {  
    static T item; // 예러  
    static int compare(T t1, T t2) { ... } // 예러  
    ...  
}
```

- 지네릭 타입의 배열 T[]를 생성하는 것은 허용되지 않는다.

```
class Box<T> {  
    T[] itemArr; // OK. T타입의 배열을 위한 참조변수  
    ...  
    T[] toArray() {  
        T[] tmpArr = new T[itemArr.length]; // 예러. 지네릭 배열 생성불가  
        ...  
        return tmpArr;  
    }  
    ...  
}
```

1.5 지네릭 클래스의 객체 생성과 사용

- 지네릭 클래스 Box<T>의 선언

```
class Box<T> {  
    ArrayList<T> list = new ArrayList<T>();  
  
    void add(T item) { list.add(item); }  
    T get(int i) { return list.get(i); }  
    ArrayList<T> getList() { return list; }  
    int size() { return list.size(); }  
    public String toString() { return list.toString(); }  
}
```

- Box<T>의 객체 생성. 참조변수와 생성자에 대입된 타입이 일치해야 함

```
Box<Apple> appleBox = new Box<Apple>(); // OK  
Box<Apple> appleBox = new Box<Grape>(); // 에러. 대입된 타입이 다르다.  
Box<Fruit> appleBox = new Box<Apple>(); // 에러. 대입된 타입이 다르다.
```

- 두 지네릭 클래스가 상속관계이고, 대입된 타입이 일치하는 것은 OK

```
Box<Apple> appleBox = new FruitBox<Apple>(); // OK. 다형성  
Box<Apple> appleBox = new Box<>(); // OK. JDK1.7부터 생략가능
```

- 대입된 타입과 다른 타입의 객체는 추가할 수 없다.

```
Box<Apple> appleBox = new Box<Apple>();  
appleBox.add(new Apple()); // OK.  
appleBox.add(new Grape()); // 에러. Box<Apple>에는 Apple객체만 추가가능
```

1.6 제한된 지네릭 클래스

- 지네릭 타입에 'extends'를 사용하면, 특정 타입의 자손들만 대입할 수 있게 제한할 수 있다.

```
class FruitBox<T extends Fruit> { // Fruit의 자손만 타입으로 지정가능
    ArrayList<T> list = new ArrayList<T>();
    void add(T item) { list.add(item); }
    ...
}
```

- `add()`의 매개변수의 타입 T도 `Fruit`와 그 자손 타입이 될 수 있다.

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
fruitBox.add(new Apple()); // OK. Apple이 Fruit의 자손
fruitBox.add(new Grape()); // OK. Grape가 Fruit의 자손
```

- 인터페이스의 경우에도 'implements'가 아닌, 'extends'를 사용

```
interface Eatable {}
class FruitBox<T extends Eatable> { ... }
class FruitBox<T extends Fruit & Eatable> { ... }
```

1.7 와일드 카드 ‘?’

- 지네릭 타입에 와일드 카드를 쓰면, 여러 타입을 대입가능
단, 와일드 카드에는 <? extends T & E>와 같이 ‘&’를 사용불가

<? extends T>	와일드 카드의 상한 제한. T와 그 자손들만 가능
<? super T>	와일드 카드의 하한 제한. T와 그 조상들만 가능
<?>	제한 없음. 모든 타입이 가능. <? extends Object>와 동일

```
static Juice makeJuice(FruitBox<? extends Fruit> box) {  
    String tmp = "";  
    for(Fruit f : box.getList()) tmp += f + " ";  
    return new Juice(tmp);  
}
```

- makeJuice()의 매개변수로 FruitBox<Apple>, FruitBox<Grape> 가능

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();  
FruitBox<Apple> appleBox = new FruitBox<Apple>();  
...  
System.out.println(Juicer.makeJuice(fruitBox)); // OK. FruitBox<Fruit>  
System.out.println(Juicer.makeJuice(appleBox)); // OK. FruitBox<Apple>
```

1.6 지네릭 메서드

- 반환타입 앞에 지네릭 타입이 선언된 메서드

```
static <T> void sort(List<T> list, Comparator<? super T> c)
```

- 클래스의 타입 매개변수<T>와 메서드의 타입 매개변수 <T>는 별개

```
class FruitBox<T> {  
    ...  
    static <T> void sort(List<T> list, Comparator<? super T> c) {  
        ...  
    }  
}
```

- 지네릭 메서드를 호출할 때, 타입 변수에 타입을 대입해야 한다.
(대부분의 경우 추정이 가능하므로 생략할 수 있음.)

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();  
FruitBox<Apple> appleBox = new FruitBox<Apple>();  
...  
System.out.println(Juicer.<Fruit>makeJuice(fruitBox));  
System.out.println(Juicer.makeJuice(appleBox)); // 대입된 타입 생략 가능
```

1.7 지네릭 타입의 형변환

- 지네릭 타입과 원시 타입간의 형변환은 가능

```
Box      box = null;
Box<Object> objBox = null;

box = (Box) objBox;           // OK. 지네릭 타입 → 원시 타입. 경고 발생
objBox = (Box<Object>) box; // OK. 원시 타입       → 지네릭 타입. 경고 발생
```

- 와일드 카드가 사용된 지네릭 타입으로는 형변환 가능

```
Box<? extends Object> wBox = new Box<String>();

FruitBox<? extends Fruit> box = null;
FruitBox<Apple> appleBox = (FruitBox<Apple>) box; // OK. 미확인 타입으로 형변환 경고
```

- <? extends Object>를 줄여서 <?>로 쓸 수 있다.

```
Optional<?> EMPTY = new Optional<?>(); // 에러. 미확인 타입의 객체는 생성불가
Optional<?> EMPTY = new Optional<Object>(); // OK.
Optional<?> EMPTY = new Optional<>();        // OK. 위의 문장과 동일
```

[주의] class Box<T extends Fruit>의 경우 Box<?> b = new Box<>; 는 Box<?> b = new Box<Fruit>; 이다.

1.8 지네릭 타입의 제거 (컴파일러의 작동 방법)

- 컴파일러는 지네릭 타입을 제거하고, 필요한 곳에 형변환을 넣는다.

① 지네릭 타입의 경계(bound)를 제거

```
class Box<T extends Fruit> {  
    void add(T t) {  
        ...  
    }  
}
```



```
class Box {  
    void add(Fruit t) {  
        ...  
    }  
}
```

② 지네릭 타입 제거 후에 타입이 불일치하면, 형변환을 추가

```
T get(int i) {  
    return list.get(i);  
}
```



```
Fruit get(int i) {  
    return (Fruit)list.get(i);  
}
```

③ 와일드 카드가 포함된 경우, 적절한 타입으로 형변환 추가

```
static Juice makeJuice(FruitBox<? extends Fruit> box) {  
    String tmp = "";  
    for(Fruit f : box.getList()) tmp += f + " ";  
    return new Juice(tmp);  
}
```



```
static Juice makeJuice(FruitBox box) {  
    String tmp = "";  
    Iterator it = box.getList().iterator();  
    while(it.hasNext()) {  
        tmp += (Fruit)it.next() + " ";  
    }  
    return new Juice(tmp);  
}
```

2. 열거형(Enums)

2.1 열거형이란?

- 관련된 상수들을 같이 묶어 놓은 것. Java는 타입에 안전한 열거형을 제공

```
class Card {  
    static final int CLOVER = 0;  
    static final int HEART = 1;  
    static final int DIAMOND = 2;  
    static final int SPADE = 3;  
  
    static final int TWO = 0;  
    static final int THREE = 1;  
    static final int FOUR = 2;  
  
    final int kind;  
    final int num;  
}
```

```
class Card {  
    enum Kind { CLOVER, HEART, DIAMOND, SPADE } // 열거형 Kind를 정의  
    enum Value { TWO, THREE, FOUR } // 열거형 Value를 정의  
  
    final Kind kind; // 타입이 int가 아닌 Kind임에 유의하자.  
    final Value value;  
}
```



2.2 열거형의 정의와 사용

- 열거형을 정의하는 방법

```
enum 열거형이름 { 상수명1, 상수명2, ... }
```

- 열거형 타입의 변수를 선언하고 사용하는 방법

```
enum Direction { EAST, SOUTH, WEST, NORTH }

class Unit {
    int x, y;      // 유닛의 위치
    Direction dir; // 열거형을 인스턴스 변수로 선언

    void init() {
        dir = Direction.EAST; // 유닛의 방향을 EAST로 초기화
    }
}
```

- 열거형 상수의 비교에 ==와 compareTo() 사용 가능

```
if(dir==Direction.EAST) {
    x++;
} else if (dir > Direction.WEST) { // 에러. 열거형 상수에 비교연산자 사용불가
    ...
} else if (dir.compareTo(Direction.WEST)>0) { // compareTo()는 가능
    ...
}
```

2.3 모든 열거형의 조상 – java.lang.Enum

- 모든 열거형은 **Enum**의 자손이며, 아래의 메서드를 상속받는다.

메서드	설명
Class<E> getDeclaringClass()	열거형의 Class객체를 반환한다.
String name()	열거형 상수의 이름을 문자열로 반환한다.
int ordinal()	열거형 상수가 정의된 순서를 반환한다.(0부터 시작)
T valueOf(Class<T> enumType, String name)	지정된 열거형에서 name과 일치하는 열거형 상수를 반환한다.

- 컴파일러가 자동적으로 추가해주는 **values()**도 있다.

```
static E values()
static E valueOf(String name)

Direction d = Direction.valueOf("WEST");
```

2.4 열거형에 멤버 추가하기

- 불연속적인 열거형 상수의 경우, 원하는 값을 괄호()안에 적는다.

```
enum Direction { EAST(1), SOUTH(5), WEST(-1), NORTH(10) }
```

- 괄호()를 사용하려면, 인스턴스 변수와 생성자를 새로 추가해 줘야 한다.

```
enum Direction {  
    EAST(1), SOUTH(5), WEST(-1), NORTH(10); // 끝에 ';'를 추가해야 한다.  
  
    private final int value; // 정수를 저장할 필드(인스턴스 변수)를 추가  
    Direction(int value) { this.value = value; } // 생성자를 추가  
  
    public int getValue() { return value; }  
}
```

- 열거형의 생성자는 목시적으로 **private**이므로, 외부에서 객체생성 불가

```
Direction d = new Direction(1); // 에러. 열거형의 생성자는 외부에서 호출불가
```

2.4 열거형의 이해

- 열거형 Direction이 아래와 같이 선언되어 있을 때,

```
enum Direction { EAST, SOUTH, WEST, NORTH }
```

- 열거형 Direction은 아래와 같은 클래스로 선언된 것과 유사하다.

```
class Direction {  
    static final Direction EAST = new Direction("EAST");  
    static final Direction SOUTH = new Direction("SOUTH");  
    static final Direction WEST = new Direction("WEST");  
    static final Direction NORTH = new Direction("NORTH");  
  
    private String name;  
  
    private Direction(String name) {  
        this.name = name;  
    }  
}
```

3. 애너테이션(Annotation) 어노테이션

3.1 애너테이션이란?

- 주석처럼 프로그래밍 언어에 영향을 미치지 않으며, 유용한 정보를 제공

```
/**  
 * The common interface extended by all annotation types. Note that an  
 * interface that manually extends this one does <i>not</i> define  
 * an annotation type. Also note that this interface does not itself  
 * define an annotation type.  
 *  
 * @author Josh Bloch  
 * @since 1.5  
 */  
public interface Annotation {  
    ...  
}
```

- 애너테이션의 사용 예

```
@Test // 이 메서드가 테스트 대상임을 테스트 프로그램에게 알린다.  
public void method() {  
    ...  
}
```

3.2 표준 애너테이션

- Java에서 제공하는 애너테이션

애너테이션	설명
@Override	컴파일러에게 오버라이딩하는 메서드라는 것을 알린다.
@Deprecated	앞으로 사용하지 않을 것을 권장하는 대상에 붙인다.
@SuppressWarnings	컴파일러의 특정 경고메시지가 나타나지 않게 해준다.
@SafeVarargs	지네릭스 타입의 가변인자에 사용한다.(JDK1.7)
@FunctionalInterface	함수형 인터페이스라는 것을 알린다.(JDK1.8)
@Native	native메서드에서 참조되는 상수 앞에 붙인다.(JDK1.8)
@Target*	애너테이션이 적용가능한 대상을 지정하는데 사용한다.
@Documented*	애너테이션 정보가 javadoc으로 작성된 문서에 포함되게 한다.
@Inherited*	애너테이션이 자손 클래스에 상속되도록 한다.
@Retention*	애너테이션이 유지되는 범위를 지정하는데 사용한다.
@Repeatable*	애너테이션을 반복해서 적용할 수 있게 한다.(JDK1.8)

▲ 표 12-2 자바에서 기본적으로 제공하는 표준 애너테이션 (*가 붙은 것은 메타 애너테이션)

3.2 표준 애너테이션 - @Override

- 오버라이딩을 올바르게 했는지 컴파일러가 체크하게 한다.
- 오버라이딩할 때 메서드 이름을 잘못 적는 실수를 하는 경우가 많다.

```
class Parent {  
    void parentMethod() { }  
}  
  
class Child extends Parent {  
    void parentmethod() { } // 오버라이딩하려 했으나 실수로 이름을 잘못적음  
}
```

- 오버라이딩 할 때는 메서드 선언부 앞에 **@Override**를 붙이자.

```
class Child extends Parent {  
    void parentmethod(){}
}
```



```
class Child extends Parent {  
    @Override  
    void parentmethod(){}
}
```

▼ 컴파일 결과

```
AnnotationEx1.java:6: error: method does not override or implement a method  
from a supertype  
    @Override  
    ^  
1 error
```

3.2 표준 애너테이션 - @Deprecated

- 앞으로 사용하지 않을 것을 권장하는 필드나 메서드에 붙인다.
- `@Deprecated`의 사용 예, `Date`클래스의 `getDate()`

```
int          getDate()
Deprecated.
As of JDK version 1.1, replaced by
Calendar.get(Calendar.DAY_OF_MONTH).

@Deprecated
public int getDate() {
    return normalize().getDayOfMonth();
}
```

- `@Deprecated`가 붙은 대상이 사용된 코드를 컴파일하면 나타나는 메시지

```
Note: AnnotationEx2.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

3.2 표준 애너테이션 - @FunctionalInterface

- 함수형 인터페이스에 붙이면, 컴파일러가 올바르게 작성했는지 체크
- 함수형 인터페이스에는 **하나의 추상메서드만** 가져야 한다는 제약이 있음

```
@FunctionalInterface  
public interface Runnable {  
    public abstract void run(); // 추상 메서드  
}
```

3.2 표준 애너테이션 - @SuppressWarnings

- 컴파일러의 경고메시지가 나타나지 않게 억제한다.
- 괄호()안에 억제하고자하는 경고의 종류를 문자열로 지정

```
@SuppressWarnings("unchecked")           // 지네릭스와 관련된 경고를 억제
ArrayList list = new ArrayList();    // 지네릭 타입을 지정하지 않았음.
list.add(obj);                      // 여기서 경고가 발생
```

- 둘 이상의 경고를 동시에 억제하려면 다음과 같이 한다.

```
@SuppressWarnings({"deprecation", "unchecked", "varargs"})
```

- '-Xlint'옵션으로 컴파일하면, 경고메시지를 확인할 수 있다.

괄호[]안이 경고의 종류. 아래의 경우 rawtypes

```
C:\jdk1.8\work\ch12>javac -Xlint AnnotationTest.java
AnnotationTest.java:15: warning: [rawtypes] found raw type: List
    public static void sort(List list) {
                           ^
missing type arguments for generic class List<E>
where E is a type-variable:
    E extends Object declared in interface List
```

3.2 표준 애너테이션 - @SafeVarargs

- 가변인자의 타입이 `non-reifiable`인 경우 발생하는 `unchecked`경고를 억제
- 생성자 또는 `static`이나 `final`이 붙은 메서드에만 붙일 수 있다.
(오버라이딩이 가능한 메서드에 사용불가)
- `@SafeVarargs`에 의한 경고의 억제를 위해 `@SuppressWarnings`을 사용

```
@SafeVarargs           // 'unchecked' 경고를 억제한다.
@SuppressWarnings("varargs") // 'varargs' 경고를 억제한다.
public static <T> List<T> asList(T... a) {
    return new ArrayList<>(a);
}
```

`non-reifiable` : 컴파일시에 타입 정보가 구체화되지 않는 것 - 경고(warning)

`reifiable` : 컴파일시에 타입 정보가 구체화되는 것

3.3 메타 애너테이션 - @Target

- 메타 애너테이션은 ‘애너테이션을 위한 애너테이션’
- 애너테이션을 정의할 때, 적용대상이나 유지기간의 지정에 사용
- **@Target**은 애너테이션을 적용할 수 있는 대상의 지정에 사용

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})  
@Retention(RetentionPolicy.SOURCE)  
public @interface SuppressWarnings {  
    String[] value();  
}
```

대상 타입	의미
ANNOTATION_TYPE	애너테이션
CONSTRUCTOR	생성자
FIELD	필드(멤버변수, enum상수)
LOCAL_VARIABLE	지역변수
METHOD	메서드
PACKAGE	패키지
PARAMETER	매개변수
TYPE	타입(클래스, 인터페이스, enum)
TYPE_PARAMETER	타입 매개변수(JDK1.8)
TYPE_USE	타입이 사용되는 모든 곳(JDK1.8)

3.3 메타 애너테이션 - @Retention

- 애너테이션이 유지(retention)되는 기간을 지정하는데 사용

유지 정책	의미
SOURCE	소스 파일에만 존재. 클래스파일에는 존재하지 않음.
CLASS	클래스 파일에 존재. 실행시에 사용불가. 기본값
RUNTIME	클래스 파일에 존재. 실행시에 사용가능.

▲ 표 12-4 애너테이션 유지정책(retention policy)의 종류

- 컴파일러에 의해 사용되는 애너테이션의 유지 정책은 SOURCE이다.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override { }
```

- 실행시에 사용 가능한 애너테이션의 정책은 RUNTIME이다.

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface { }
```

3.3 메타 애너테이션 - @Documented, @Inherited

- javadoc으로 작성한 문서에 포함시키려면 **@Documented**를 붙인다.

```
@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface FunctionalInterface {}
```

- 애너테이션을 자손 클래스에 상속하고자 할 때, **@Inherited**를 붙인다.

```
@Inherited // @SupperAnno가 자손까지 영향 미치게  
@interface SupperAnno {}  
  
@SupperAnno  
class Parent {}  
  
class Child extends Parent {} // Child에 애너테이션이 붙은 것으로 인식
```

3.3 메타 애너테이션 - @Repeatable

- 반복해서 붙일 수 있는 애너테이션을 정의할 때 사용

```
@Repeatable(Todos.class) // ToDo애너테이션을 여러 번 반복해서 쓸 수 있게 한다.  
@interface ToDo {  
    String value();  
}
```

- @Repeatable이 붙은 애너테이션은 반복해서 붙일 수 있다.

```
@ToDo("delete test codes.")  
@ToDo("override inherited methods")  
class MyClass {  
    ...  
}
```

3.3 메타 애너테이션 - @Native

- native메서드에 의해 참조되는 상수에 붙이는 애너테이션

```
@Native public static final long MIN_VALUE = 0x8000000000000000L;
```

- native메서드에 JVM이 설치된 OS의 메서드이다.

```
public class Object {  
    private static native void registerNatives(); // 네이티브 메서드  
  
    static {  
        registerNatives(); // 네이티브 메서드를 호출  
    }  
  
    protected native Object clone() throws CloneNotSupportedException;  
    public final native Class<?> getClass();  
    public final native void notify();  
    public final native void notifyAll();  
    public final native void wait(long timeout) throws InterruptedException;  
    public native int hashCode();  
    ...  
}
```

3.4 애너테이션 타입 정의하기

- 애너테이션을 직접 만들어 쓸 수 있다.

```
@interface 애너테이션이름 {  
    타입 요소이름(); // 애너테이션의 요소를 선언한다.  
    ...  
}
```

- 애너테이션의 메서드는 추상메서드이며, 애너테이션을 적용할 때 모두 지정해야 한다.(순서 상관없음)

```
@interface TestInfo {  
    int      count();  
    String   testedBy();  
    String[] testTools();  
    TestType testType(); // enum TestType { FIRST, FINAL }  
    DateTime testDate(); // 자신이 아닌 다른 애너테이션 (@DateTime) 을 포함할 수 있다.  
}  
  
@interface DateTime {  
    String  yymmdd();  
    String  hhmmss();  
}
```

3.5 애너테이션 요소의 기본값

- 적용시 값을 지정하지 않으면, 사용될 수 있는 기본값 지정 가능(null 제외)

```
@interface TestInfo {  
    int count() default 1;      // 기본값을 1로 지정  
}  
  
@TestInfo // @TestInfo(count=1)과 동일  
public class NewClass { ... }
```

- 요소가 하나일 때는 요소의 이름 생략 가능

```
@TestInfo(5) // @TestInfo(count=5)와 동일  
public class NewClass { ... }
```

- 요소의 타입이 배열인 경우, 괄호{}를 사용해야 한다.

```
@interface TestInfo {  
    String[] info() default {"aaa", "bbb"}; // 기본값이 여러 개인 경우. 괄호{} 사용  
    String[] info2() default "ccc"; // 기본값이 하나인 경우. 괄호 생략 가능  
}  
  
@TestInfo          // @TestInfo(info={"aaa", "bbb"}, info2="ccc")와 동일  
@TestInfo(info2={}) // @TestInfo(info={"aaa", "bbb"}, info2={})와 동일  
public class NewClass { ... }
```

3.6 모든 애너테이션의 조상 – java.lang.annotation.Annotation

- Annotation은 모든 애너테이션의 조상이지만 상속은 불가

```
@interface TestInfo extends Annotation { // 여러. 허용되지 않는 표현
    int      count();
    String   testedBy();
    ...
}
```

- 사실 Annotation은 인터페이스로 정의되어 있다.

```
package java.lang.annotation;

public interface Annotation { // Annotation 자신은 인터페이스이다.
    boolean equals(Object obj);
    int     hashCode();
    String  toString();

    Class<? extends Annotation> annotationType(); // 애너테이션의 타입을 반환
}
```

3.7 마커 애너테이션 - Marker Annotation

- 요소가 하나도 정의되지 않은 애너테이션

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {} // 마커 애너테이션. 정의된 요소가 하나도 없다.
```

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Test {} // 마커 애너테이션. 정의된 요소가 하나도 없다.
```

3.8 애너테이션 요소의 규칙

- 애너테이션의 요소를 선언할 때 아래의 규칙을 반드시 지켜야 한다.

- 요소의 타입은 기본형, String, enum, 애너테이션, Class만 허용됨
- 괄호()안에 매개변수를 선언할 수 없다.
- 예외를 선언할 수 없다.
- 요소를 타입 매개변수로 정의할 수 없다.

- 아래의 코드에서 잘못된 부분은 무엇인지 생각해보자.

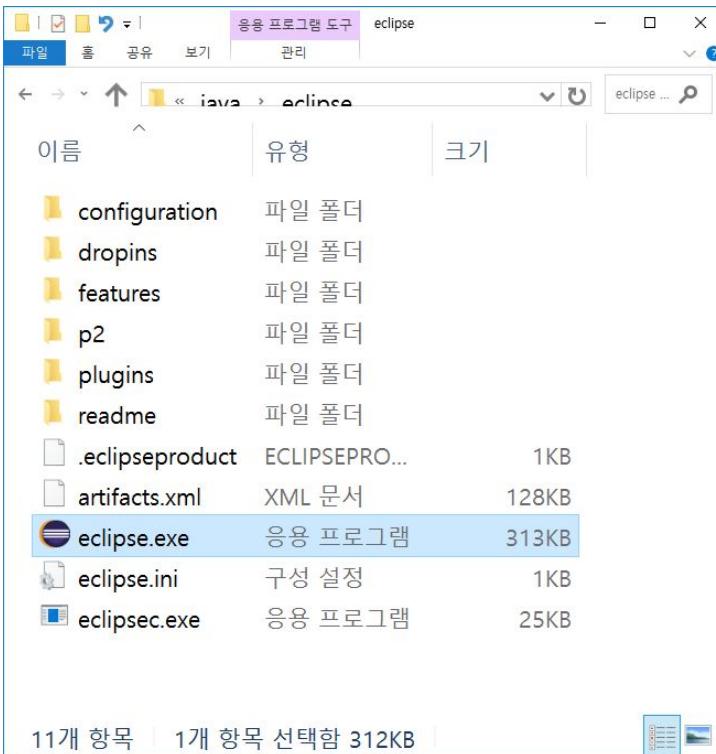
```
@interface AnnoTest {  
    int id = 100;                                // OK. 상수 선언. static final int id = 100;  
    String major(int i, int j);                  // 에러. 매개변수를 선언할 수 없음  
    String minor() throws Exception;            // 에러. 예외를 선언할 수 없음  
    ArrayList<T> list();                        // 에러. 요소의 타입에 타입 매개변수 사용불가  
}
```

쓰레드(thread)

1.1 프로세스와 쓰레드(process & thread) (1/2)

프로그램 실행 → 프로세스

▶ 프로그램 : 실행 가능한 파일(HDD, SSD)



▶ 프로세스 : 실행 중인 프로그램(메모리)

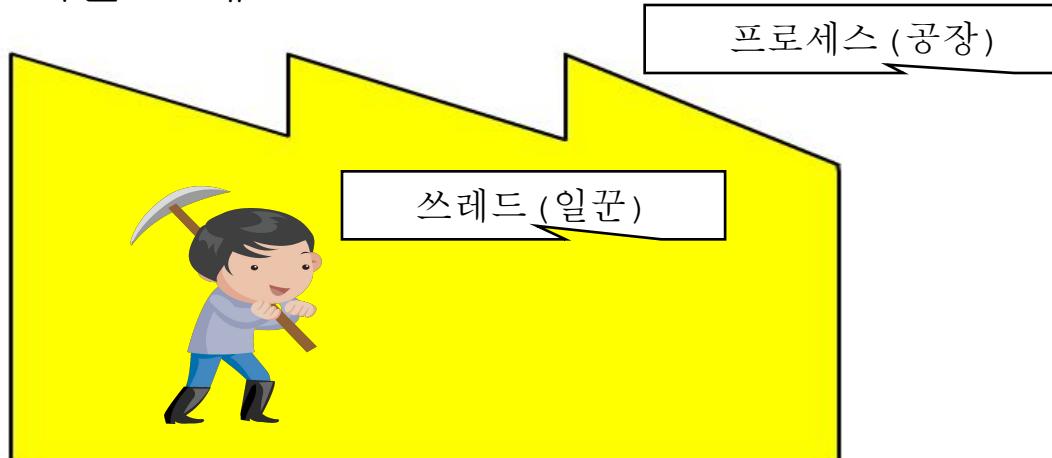
이름	45% CPU	64% 메모리
앱 (4)		
> eclipse.exe	0%	555.8MB
> Windows 탐색기(2)	0%	58.4MB
설정	0%	12.1MB
> 작업 관리자	0.2%	10.3MB
백그라운드 프로세스 (61)		
AhnLab Safe Transaction ...	7.3%	1.9MB
AhnLab Safe Transaction ...	8.6%	1.0MB

1.1 프로세스와 쓰레드(process & thread) (2/2)

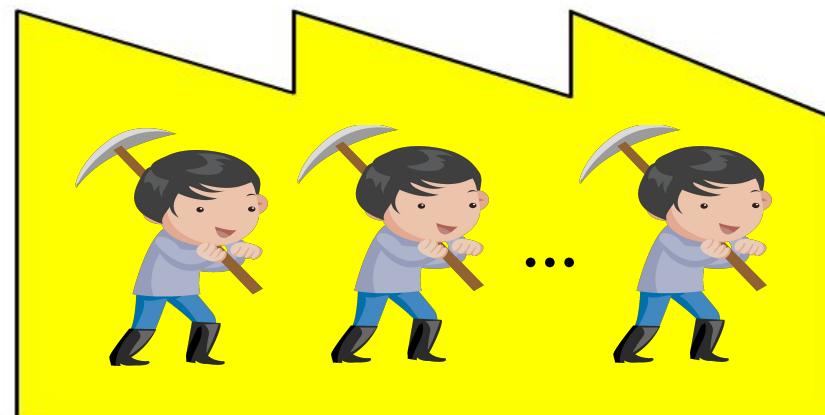
- ▶ 프로세스 : 실행 중인 프로그램, 자원(resources)과 쓰레드로 구성
- ▶ 쓰레드 : 프로세스 내에서 실제 작업을 수행.
모든 프로세스는 최소한 하나의 쓰레드를 가지고 있다.

프로세스 : 쓰레드 = 공장 : 일꾼

- ▶ 싱글 쓰레드 프로세스
= 자원+쓰레드



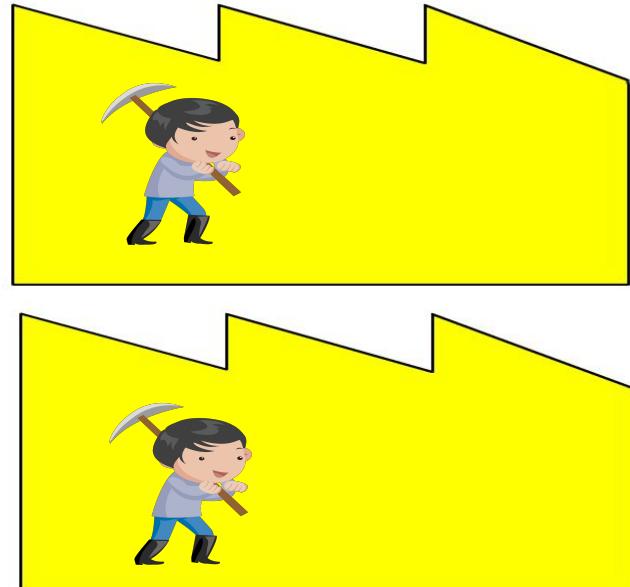
- ▶ 멀티 쓰레드 프로세스
= 자원+쓰레드+쓰레드+...+쓰레드



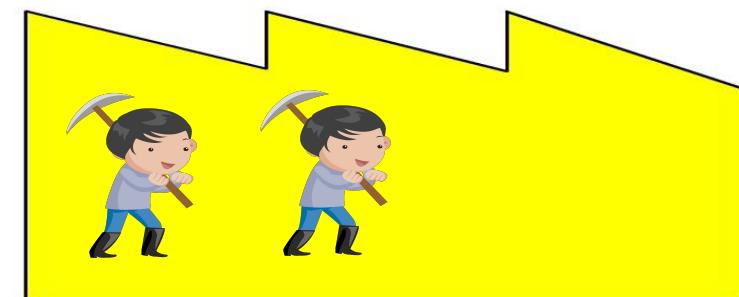
1.2 멀티프로세스 vs. 멀티쓰레드

- ▶ 멀티 태스킹(멀티 프로세싱) : 동시에 여러 프로세스를 실행시키는 것
- ▶ 멀티 쓰레딩 : 하나의 프로세스 내에 동시에 여러 쓰레드를 실행시키는 것
 - 프로세스를 생성하는 것보다 쓰레드를 생성하는 비용이 적다.
 - 같은 프로세스 내의 쓰레드들은 서로 자원을 공유한다.

2 프로세스 1 쓰레드



1 프로세스 2 쓰레드



VS.

1.3 멀티쓰레드의 장단점

대부분의 프로그램이 멀티쓰레드로 작성되어 있다.

그러나, 멀티쓰레드 프로그래밍이 장점만 있는 것은 아니다.

장점	<ul style="list-style-type: none">- 시스템 자원을 보다 효율적으로 사용할 수 있다.- 사용자에 대한 응답성(responsiveness)이 향상 된다.- 작업이 분리되어 코드가 간결해 진다. <p style="text-align: center;">“여러 모로 좋다.”</p>
단점	<ul style="list-style-type: none">- 동기화(synchronization)에 주의해야 한다.- 교착상태(dead-lock)가 발생하지 않도록 주의해야 한다.- 각 쓰레드가 효율적으로 고르게 실행될 수 있게 해야 한다. <p style="text-align: center;">“프로그래밍할 때 고려해야 할 사항들이 많다.”</p>

1.4 쓰레드의 구현과 실행

① Thread 클래스를 상속

```
class MyThread extends Thread {  
    public void run() { // Thread클래스의 run ()을 오버라이딩  
        /* 작업내용 */  
    }  
}
```

② Runnable 인터페이스를 구현

```
class MyThread2 implements Runnable {  
    public void run() { // Runnable인터페이스의 추상메서드 run ()을 구현  
        /* 작업내용 */  
    }  
}
```

```
public interface Runnable {  
    public abstract void run();  
}
```

```
MyThread t1 = new MyThread(); // 쓰레드의 생성  
t1.start(); // 쓰레드의 실행
```

```
Runnable r = new MyThread2();  
Thread t2 = new Thread(r); // Thread(Runnable r)  
// Thread t2 = new Thread(new MyThread2());  
t2.start();
```

1.5 start()와 run()

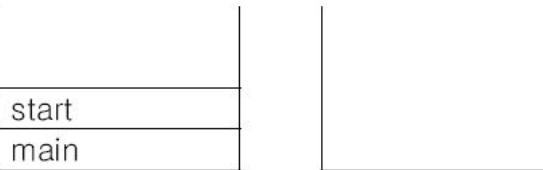
```
class ThreadTest {  
    public static void main(String args[]) {  
        MyThread t1 = new MyThread();  
        t1.start();  
    }  
}
```

```
class MyThread extends Thread {  
    public void run() {  
        //...  
    }  
}
```

1. Call stack



2. Call stack



3. Call stack



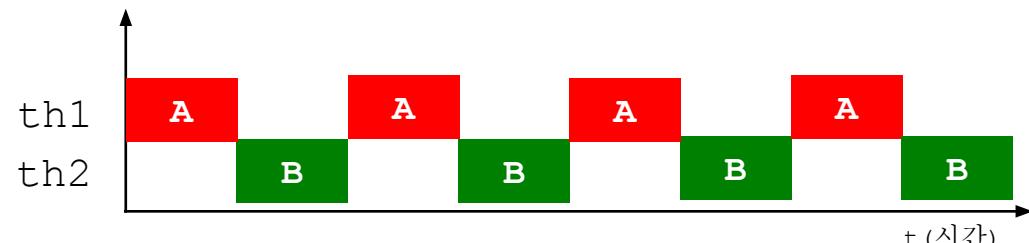
4. Call stack



2.1 싱글쓰레드 vs. 멀티쓰레드(1/3)

▶ 싱글쓰레드

```
class ThreadTest {  
    public static void main(String args[]) {  
        for(int i=0;i<300;i++) {  
            System.out.println("-");  
        }  
  
        for(int i=0;i<300;i++) {  
            System.out.println(" | ");  
        } // main  
    } // main
```



▶ 멀티쓰레드

```
class ThreadTest {  
    public static void main(String args[]) {  
        MyThread1 th1 = new MyThread1();  
        MyThread2 th2 = new MyThread2();  
        th1.start();  
        th2.start();  
    }  
  
    class MyThread1 extends Thread {  
        public void run() {  
            for(int i=0;i<300;i++) {  
                System.out.println("-");  
            } // run()  
        }  
  
    class MyThread2 extends Thread {  
        public void run() {  
            for(int i=0;i<300;i++) {  
                System.out.println(" | ");  
            } // run()  
        }
```

2.1 싱글쓰레드 vs. 멀티쓰레드(2/3) – 병행과 병렬

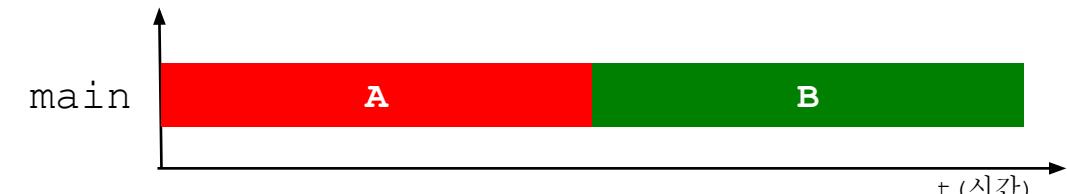
▶ 멀티쓰레드

```
class ThreadTest {
    public static void main(String args[]) {
        MyThread1 th1 = new MyThread1();
        MyThread2 th2 = new MyThread2();
        th1.start();
        th2.start();
    }
}

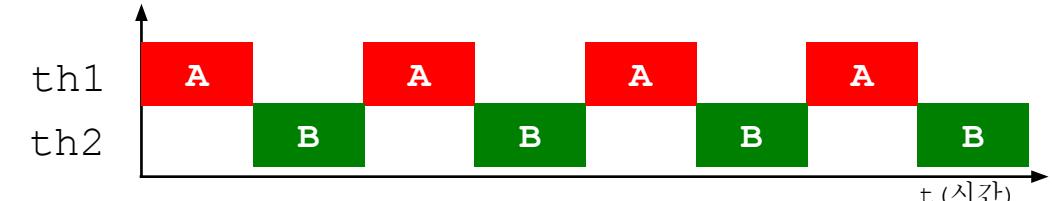
class MyThread1 extends Thread {
    public void run() {
        for(int i=0;i<300;i++) {
            System.out.println("-");
        }
    } // run()
}

class MyThread2 extends Thread {
    public void run() {
        for(int i=0;i<300;i++) {
            System.out.println("=");
        }
    } // run()
}
```

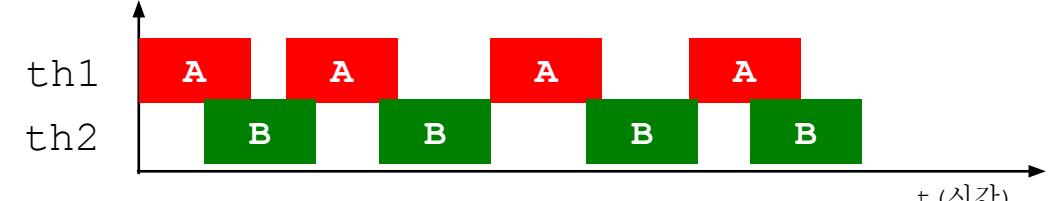
▶ 싱글 코어 – 순차 실행



▶ 싱글 코어 – 병행(concurrent)



▶ 멀티 코어 – 병행(concurrent)



▶ 멀티 코어 – 병렬(parallel)



2.1 싱글쓰레드 vs. 멀티쓰레드(3/3) - blocking

```
class ThreadEx6 {
    public static void main(String[] args) {
        String input = JOptionPane.showInputDialog("아무 값이나 입력하세요.");
        System.out.println("입력하신 값은 " + input + "입니다.");

        for(int i=10; i > 0; i--) {
            System.out.println(i);
            try { Thread.sleep(1000); } catch(Exception e) {}
        }
    } // main
}
```

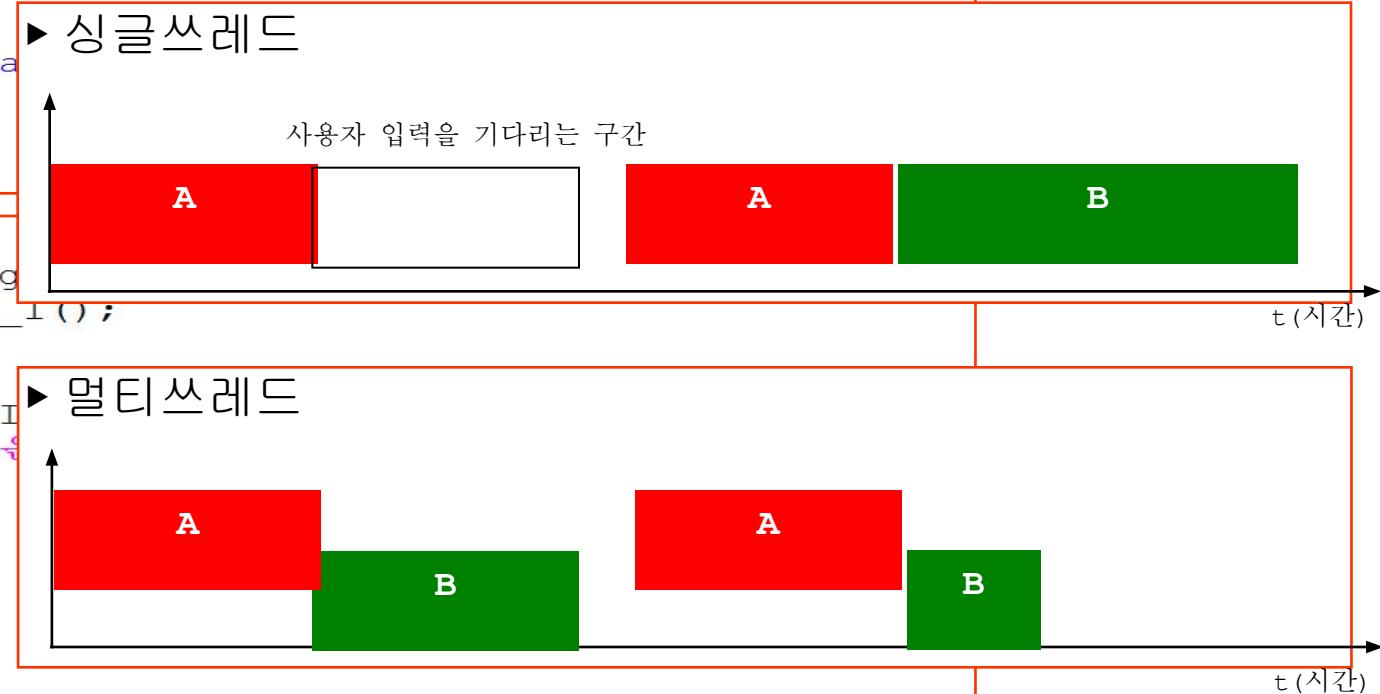
```
class ThreadEx7 {
    public static void main(String[] args) {
        ThreadEx7_1 th1 = new ThreadEx7_1();
        th1.start();

        String input = JOptionPane.showInputDialog("아무 값이나 입력하세요.");
        System.out.println("입력하신 값은 " + input + "입니다.");
    }
}

class ThreadEx7_1 extends Thread {
    public void run() {
        for(int i=10; i > 0; i--) {
            System.out.println(i);
            try { sleep(1000); } catch(Exception e) {}
        }
    } // run()
}
```

2.1 싱글쓰레드 vs. 멀티쓰레드(3/3) - blocking

```
class ThreadEx6 {  
    public static void main(String[] args){  
        String input = JOptionPane.showInputDialog("아무 값이나 입력하세요.");  
        System.out.println("입력하신 값은 " + input + "입니다.");  
  
        for(int i=10; i > 0; i--) {  
            System.out.println(i);  
            try { Thread.sleep(1000); } catch (Exception e) {}  
        }  
    } // main  
}  
  
class ThreadEx7 {  
    public static void main(String[] args){  
        ThreadEx7_1 th1 = new ThreadEx7_1();  
        th1.start();  
  
        String input = JOptionPane.showInputDialog("입력하신 값은");  
        System.out.println("입력하신 값은");  
    }  
  
    class ThreadEx7_1 extends Thread {  
        public void run() {  
            for(int i=10; i > 0; i--) {  
                System.out.println(i);  
                try { sleep(1000); } catch(Exception e) {}  
            }  
        } // run()  
    } // ThreadEx7
```



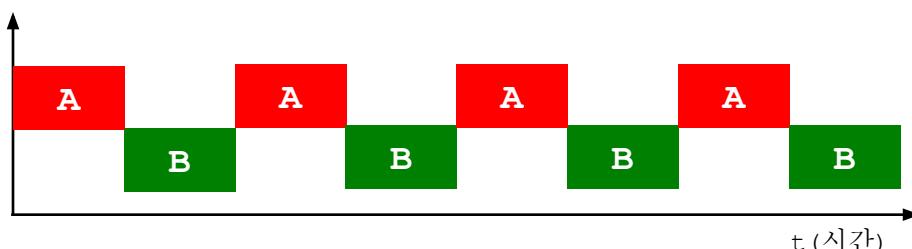
2.2 쓰레드의 우선순위(priority of thread)

- 작업의 중요도에 따라 쓰레드의 우선순위를 다르게 하여 특정 쓰레드가 더 많은 작업시간을 갖게 할 수 있다.

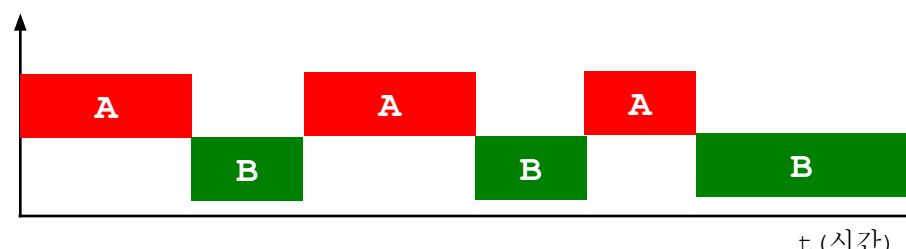
```
void setPriority(int newPriority)    쓰레드의 우선순위를 지정한 값으로 변경한다.  
int  getPriority()                  쓰레드의 우선순위를 반환한다.
```

```
public static final int MAX_PRIORITY  = 10 // 최대우선순위  
public static final int MIN_PRIORITY  = 1  // 최소우선순위  
public static final int NORM_PRIORITY = 5  // 보통우선순위
```

▶ 우선순위가 같은 경우



▶ A의 우선순위가 높은 경우



2.3 쓰레드 그룹(ThreadGroup)

`Thread(ThreadGroup group, String name)`

`Thread(ThreadGroup group, Runnable target)`

`Thread(ThreadGroup group, Runnable target, String name)`

`Thread(ThreadGroup group, Runnable target, String name, long stackSize)`

- 서로 관련된 쓰레드를 그룹으로 묶어서 다루기 위한 것(보안상의 이유)
- 모든 쓰레드는 반드시 하나의 쓰레드 그룹에 포함되어 있어야 한다.
- 쓰레드 그룹을 지정하지 않고 생성한 쓰레드는 ‘main쓰레드 그룹’에 속한다.
- 자신을 생성한 쓰레드(부모 쓰레드)의 그룹과 우선순위를 상속받는다.

생성자 / 메서드	설명
<code>ThreadGroup(String name)</code>	지정된 이름의 새로운 쓰레드 그룹을 생성
<code>ThreadGroup(ThreadGroup parent, String name)</code>	지정된 쓰레드 그룹에 포함되는 새로운 쓰레드 그룹을 생성
<code>int activeCount()</code>	쓰레드 그룹에 포함된 활성상태에 있는 쓰레드의 수를 반환
<code>int activeGroupCount()</code>	쓰레드 그룹에 포함된 활성상태에 있는 쓰레드 그룹의 수를 반환
<code>void checkAccess()</code>	현재 실행중인 쓰레드가 쓰레드 그룹을 변경할 권한이 있는지 체크.
<code>void destroy()</code>	쓰레드 그룹과 하위 쓰레드 그룹까지 모두 삭제한다.
<code>int enumerate(Thread[] list)</code> <code>int enumerate(Thread[] list, boolean recurse)</code> <code>int enumerate(ThreadGroup[] list)</code> <code>int enumerate(ThreadGroup[] list, boolean recurse)</code>	쓰레드 그룹에 속한 쓰레드 또는 하위 쓰레드 그룹의 목록을 지정된 배열에 담고 그 개수를 반환. 두 번째 매개변수인 recurse의 값을 true로 하면 쓰레드 그룹에 속한 하위 쓰레드 그룹에 쓰레드 또는 쓰레드 그룹까지 배열에 담는다.
<code>int getMaxPriority()</code>	쓰레드 그룹의 최대우선순위를 반환
<code>String getName()</code>	쓰레드 그룹의 이름을 반환
<code>ThreadGroup getParent()</code>	쓰레드 그룹의 상위 쓰레드그룹을 반환
<code>void interrupt()</code>	쓰레드 그룹에 속한 모든 쓰레드를 interrupt
<code>boolean isDaemon()</code>	쓰레드 그룹이 데몬 쓰레드그룹인지 확인
<code>boolean isDestroyed()</code>	쓰레드 그룹이 삭제되었는지 확인
<code>void list()</code>	쓰레드 그룹에 속한 쓰레드와 하위 쓰레드그룹에 대한 정보를 출력
<code>boolean parentOf(ThreadGroup g)</code>	지정된 쓰레드 그룹의 상위 쓰레드그룹인지 확인
<code>void setDaemon(boolean daemon)</code>	쓰레드 그룹을 데몬 쓰레드그룹으로 설정/해제
<code>void setMaxPriority(int pri)</code>	쓰레드 그룹의 최대우선순위를 설정

2.4 데몬 쓰레드(daemon thread)

- 일반 쓰레드(non-daemon thread)의 작업을 돋는 보조적인 역할을 수행.
- 일반 쓰레드가 모두 종료되면 자동적으로 종료된다.
- 가비지 컬렉터, 자동저장, 화면자동갱신 등에 사용된다.
- 무한루프와 조건문을 이용해서 실행 후 대기하다가 특정조건이 만족되면 작업을 수행하고 다시 대기하도록 작성한다.

boolean isDaemon() - 쓰레드가 데몬 쓰레드인지 확인한다. 데몬 쓰레드이면 true를 반환

void setDaemon(boolean on) - 쓰레드를 데몬 쓰레드로 또는 사용자 쓰레드로 변경
매개변수 on을 true로 지정하면 데몬 쓰레드가 된다.

* **setDaemon(boolean on)**은 반드시
start()를 호출하기 전에
실행되어야 한다.

그렇지 않으면
IllegalThreadStateException이
발생한다.

```
public void run() {  
    while(true) {  
        try {  
            Thread.sleep(3 * 1000); // 3초마다  
        } catch(InterruptedException e) {}  
  
        // autoSave의 값이 true이면 autoSave()를 호출한다.  
        if(autoSave) {  
            autoSave();  
        }  
    }  
}
```

3.1 쓰레드의 실행 제어

- 쓰레드의 실행을 제어(스케줄링)할 수 있는 메서드가 제공된다.

이들을 활용해서 보다 효율적인 프로그램의 작성할 수 있다.

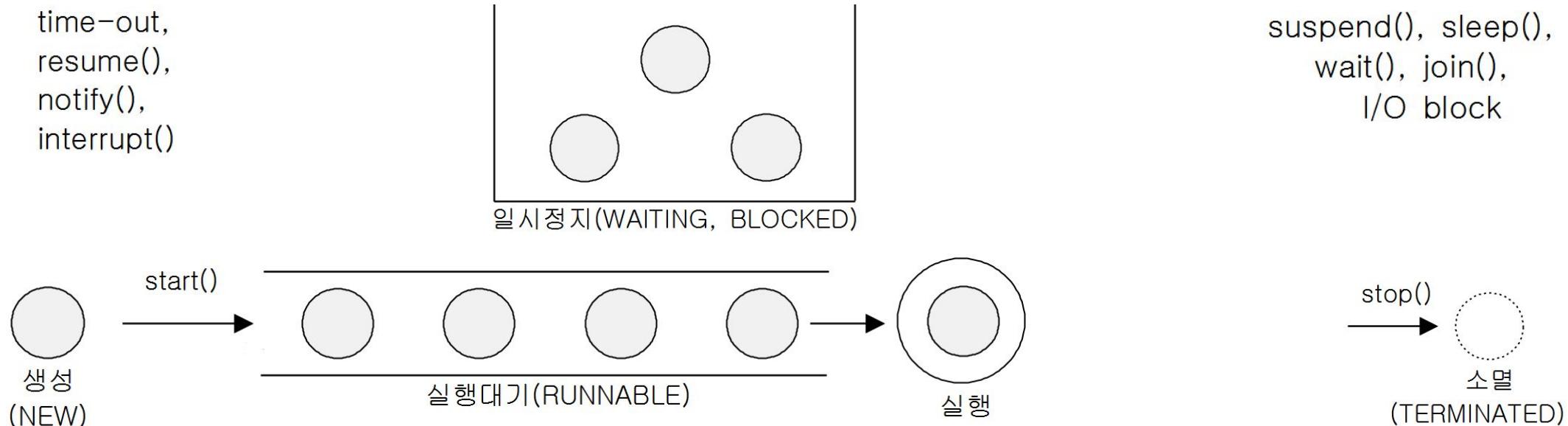
메서드	설명
static void sleep(long millis) static void sleep(long millis, int nanos)	지정된 시간(천분의 일초 단위)동안 쓰레드를 일시정지시킨다. 지정한 시간이 지나고 나면, 자동적으로 다시 실행대기상태가 된다.
void join() void join(long millis) void join(long millis, int nanos)	지정된 시간동안 쓰레드가 실행되도록 한다. 지정된 시간이 지나거나 작업이 종료되면 join()을 호출한 쓰레드로 다시 돌아와 실행을 계속한다.
void interrupt()	sleep()이나 join()에 의해 일시정지상태인 쓰레드를 깨워서 실행대기상태로 만든다. 해당 쓰레드에서는 InterruptedException이 발생함으로써 일시정지 상태를 벗어나게 된다.
void stop()	쓰레드를 즉시 종료시킨다.
void suspend()	쓰레드를 일시정지시킨다. resume()을 호출하면 다시 실행대기상태가 된다.
void resume()	suspend()에 의해 일시정지상태에 있는 쓰레드를 실행대기상태로 만든다.
static void yield()	실행 중에 자신에게 주어진 실행시간을 다른 쓰레드에게 양보(yield)하고 자신은 실행대기상태가 된다.

▲ 표 13-2 쓰레드의 스케줄링과 관련된 메서드

* resume(), stop(), suspend()는 쓰레드를 교착상태로 만들기 쉽기 때문에 deprecated되었다.

3.2 쓰레드의 상태(state of thread)

상태	설명
NEW	쓰레드가 생성되고 아직 start()가 호출되지 않은 상태
RUNNABLE	실행 중 또는 실행 가능한 상태
BLOCKED	동기화블럭에 의해서 일시정지된 상태 (lock이 풀릴 때까지 기다리는 상태)
WAITING, TIMED_WAITING	쓰레드의 작업이 종료되지는 않았지만 실행가능하지 않은(unrunnable) 일시정지 상태. TIMED_WAITING은 일시정지시간이 지정된 경우를 의미한다.
TERMINATED	쓰레드의 작업이 종료된 상태



3.3 쓰레드의 실행제어 메서드(1/5) – sleep()

- 현재 쓰레드를 지정된 시간동안 멈추게 한다.

```
static void sleep(long millis)           // 천분의 일초 단위  
static void sleep(long millis, int nanos) // 천분의 일초 + 나노초
```

- 예외처리를 해야 한다.(InterruptedException이 발생하면 깨어남)

```
try {  
    Thread.sleep(1, 500000); // 쓰레드를 0.0015초 동안 멈추게 한다.  
} catch(InterruptedException e) {}
```

```
void delay(long millis) {  
    try {  
        Thread.sleep(millis);  
    } catch(InterruptedException e) {}  
}
```

- 특정 쓰레드를 지정해서 멈추게 하는 것은 불가능하다.

```
try {  
    th1.sleep(2000);  
} catch(InterruptedException e) {}
```

```
try {  
    Thread.sleep(2000);  
} catch(InterruptedException e) {}
```

3.3 쓰레드의 실행제어 메서드(2/5) – interrupt()

- 대기상태(WAITING)인 쓰레드를 실행대기 상태(RUNNABLE)로 만든다.

```
void      interrupt()
```

쓰레드의 interrupted상태를 false에서 true로 변경.

```
boolean  isInterrupted()
```

쓰레드의 interrupted상태를 반환.

```
static boolean interrupted()
```

현재 쓰레드의 interrupted상태를 알려주고, false로 초기화

```
public static void main(String[] args){  
    ThreadEx13_2 th1 = new ThreadEx13_2();  
    th1.start();  
  
    ...  
    th1.interrupt(); // interrupt()를 호출하면, interrupted상태가 true가 된다.  
    ...  
    System.out.println("isInterrupted():"+ th1.isInterrupted()); // true  
}
```

```
class Thread { // 알기 쉽게 변경한 코드  
    ...  
    boolean interrupted = false;  
    ...  
    boolean isInterrupted() {  
        return interrupted;  
    }  
  
    boolean interrupt() {  
        interrupted = true;  
    }  
}
```

```
class ThreadEx13_2 extends Thread {  
    public void run() {  
        ...  
        while( downloaded && !isInterrupted() ) {  
            // download를 수행한다.  
            ...  
        }  
        System.out.println("다운로드가 끝났습니다.");  
    } // main  
}
```

3.3 쓰레드의 실행제어 메서드(3/5) -

**suspend(), resume()
, stop()**

- 쓰레드의 실행을 일시정지, 재개, 완전정지 시킨다. 교착상태에 빠지기 쉽다.

void suspend() 쓰레드를 일시정지 시킨다.

void resume() suspend()에 의해 일시정지된 쓰레드를 실행대기상태로 만든다.

void stop() 쓰레드를 즉시 종료시킨다.

- suspend(), resume(), stop()은 deprecated되었으므로, 직접 구현해야 한다.

```
class ThreadEx17_1 implements Runnable {
    boolean suspended = false;
    boolean stopped = false;

    public void run() {
        while(!stopped) {
            if(!suspended) {
                /* 쓰레드가 수행할 코드를 작성 */
            }
        }
    }
    public void suspend() { suspended = true; }
    public void resume() { suspended = false; }
    public void stop() { stopped = true; }
}
```

3.3 쓰레드의 실행제어 메서드(4/5) – yield()

- 남은 시간을 다음 쓰레드에게 양보하고, 자신(현재 쓰레드)은 실행대기한다.
- yield()와 interrupt()를 적절히 사용하면, 응답성과 효율을 높일 수 있다.

```
class MyThreadEx18 implements Runnable {  
    boolean suspended = false;  
    boolean stopped = false;  
  
    Thread th;  
  
    MyThreadEx18(String name) {  
        th = new Thread(this, name);  
    }  
  
    public void run() {  
        while(!stopped) {  
            if(!suspended) {  
                /*  
                 * 작업수행  
                */  
                try {  
                    Thread.sleep(1000);  
                } catch(InterruptedException e) {}  
            } else {  
                Thread.yield();  
            } // if  
        } // while  
    }  
  
    public void start() {  
        th.start();  
    }  
  
    public void resume() {  
        suspended = false;  
    }  
  
    public void suspend() {  
        suspended = true;  
        th.interrupt();  
    }  
  
    public void stop() {  
        stopped = true;  
        th.interrupt();  
    }  
}
```

3.3 쓰레드의 실행제어 메서드(5/5) – join()

- 지정된 시간동안 특정 쓰레드가 작업하는 것을 기다린다.

```
void join()                                // 작업이 모두 끝날 때까지
void join(long millis)                      // 천분의 일초 동안
void join(long millis, int nanos)           // 천분의 일초 + 나노초 동안
```

- 예외처리를 해야 한다.(InterruptedException이 발생하면 작업 재개)

```
public static void main(String args[]) {
    ThreadEx19_1 th1 = new ThreadEx19_1();
    ThreadEx19_2 th2 = new ThreadEx19_2();
    th1.start();
    th2.start();
    startTime = System.currentTimeMillis();

    try {
        th1.join(); // main쓰레드가 th1의 작업이 끝날 때까지 기다린다.
        th2.join(); // main쓰레드가 th2의 작업이 끝날 때까지 기다린다.
    } catch(InterruptedException e) {}

    System.out.print("소요시간:" + (System.currentTimeMillis()
        - ThreadEx19.startTime));
} // main
```

3.4 쓰레드의 실행제어 예제 – join() & interrupt()

```
public void run() {
    while(true) {
        try {
            Thread.sleep(10 * 1000); // 10초를 기다린다.
        } catch(InterruptedException e) {
            System.out.println("Awaken by interrupt().");
        }

        gc(); // garbage collection을 수행한다.
        System.out.println("Garbage Collected. Free Memory :" + freeMemory());
    }
}

for(int i=0; i < 20; i++) {
    requiredMemory = (int)(Math.random() * 10) * 20;
    // 필요한 메모리가 사용할 수 있는 양보다 적거나 전체 메모리의 60%이상 사용했을 경우 gc를 깨운다.
    if(gc.freeMemory() < requiredMemory ||
       gc.freeMemory() < gc.totalMemory() * 0.4)
    {
        gc.interrupt(); // 잠자고 있는 쓰레드 gc를 깨운다.

        //

    }
    gc.usedMemory += requiredMemory;
    System.out.println("usedMemory:" + gc.usedMemory);
}
```

1.13 쓰레드의 동기화 - synchronized

- 한 번에 하나의 쓰레드만 객체에 접근할 수 있도록 객체에 락(lock)을 걸어서 데이터의 일관성을 유지하는 것.

1. 특정한 객체에 lock을 걸고자 할 때

```
synchronized(객체의 참조변수) {  
    //...  
}
```

2. 메서드에 lock을 걸고자 할 때

```
public synchronized void calcSum() {  
    //...  
}
```

```
public synchronized void withdraw(int money) {  
    if(balance >= money) {  
        try {  
            Thread.sleep(1000);  
        } catch(Exception e) {}  
  
        balance -= money;  
    }  
}
```

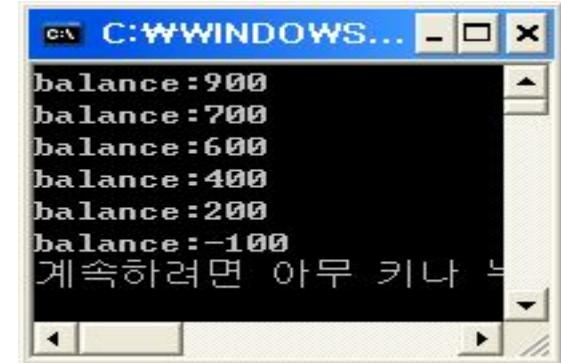
```
public void withdraw(int money) {  
    synchronized(this) {  
        if(balance >= money) {  
            try {  
                Thread.sleep(1000);  
            } catch(Exception e) {}  
  
            balance -= money;  
        } // synchronized(this)  
    }  
}
```

1.13 쓰레드의 동기화 - Example

```
class Account2 {  
    private int balance = 1000; // private으로 해야 동기화가 의미가 있다.  
  
    public int getBalance() {  
        return balance;  
    }  
  
    public synchronized void withdraw(int money){ // synchronized로 메서드를 동기화  
        if(balance >= money) {  
            try { Thread.sleep(1000); } catch(InterruptedException e) {}  
            balance -= money;  
        }  
    } // withdraw  
}  
  
class RunnableEx22 implements Runnable {  
    Account2 acc = new Account2();  
  
    public void run() {  
        while(acc.getBalance() > 0) {  
            // 100, 200, 300중의 한 값을 임으로 선택해서 출금(withdraw)  
            int money = (int)(Math.random() * 3 + 1) * 100;  
            acc.withdraw(money);  
            System.out.println("balance:"+acc.getBalance());  
        }  
    } // run()  
}
```

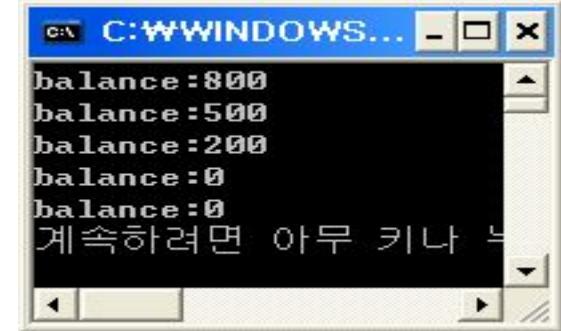
```
class ThreadEx22 {  
    public static void main(String args[]) {  
        Runnable r = new RunnableEx22();  
        new Thread(r).start();  
        new Thread(r).start();  
    }  
}
```

▶synchronized 없을 때



balance:900
balance:700
balance:600
balance:400
balance:200
balance:-100
계속하려면 아무 키나 누르세요.

▶synchronized 있을 때



balance:800
balance:500
balance:200
balance:0
balance:0
계속하려면 아무 키나 누르세요.

1.14 쓰레드의 동기화 – wait(), notify(), notifyAll()

- 동기화의 효율을 높이기 위해 `wait()`, `notify()`를 사용.
- `Object`클래스에 정의되어 있으며, 동기화 블록 내에서만 사용할 수 있다.

- `wait()` – 객체의 lock을 풀고 쓰레드를 해당 객체의 waiting pool에 넣는다.
- `notify()` – waiting pool에서 대기중인 쓰레드 중의 하나를 깨운다.
- `notifyAll()` – waiting pool에서 대기중인 모든 쓰레드를 깨운다.

```
class Account {  
    int balance = 1000;  
  
    public synchronized void withdraw(int money) {  
        while(balance < money) {  
            try {  
                wait(); // 대기 – 락을 풀고 기다린다. 통지를 받으면 락을 재획득 (ReEntrance)  
            } catch(InterruptedException e) {}  
        }  
  
        balance -= money;  
    } // withdraw  
  
    public synchronized void deposit(int money) {  
        balance += money;  
        notify(); // 통지 – 대기중인 쓰레드 중 하나에게 알림.  
    }  
}
```

생산자-소비자 문제

생산자-소비자 문제(producer-consumer problem)^{[1][2]}는 여러 개의 프로세스를 어떻게 동기화할 것인가에 관한 고전적인 문제이다. 한정 버퍼 문제(bounded-buffer problem)라고도 한다.

유한한 개수의 물건(데이터)을 임시로 보관하는 보관함(버퍼)에 여러 명의 생산자들과 소비자들이 접근한다. 생산자는 물건이 하나 만들어지면 그 공간에 저장한다. 이때 저장할 공간이 없는 문제가 발생할 수 있다. 소비자는 물건이 필요할 때 보관함에서 물건을 하나 가져온다. 이 때는 소비할 물건이 없는 문제가 발생할 수 있다.

이 문제를 해결하는 것을 생산자-소비자 협동이라 하며, 버퍼가 동기화되어 정상적으로 동작하는 상태를 뜻한다. 문제를 해결하기 위해 세마포어를 활용할 수 있다.

From : WIKI

- 생산자 프로세스

```
do {  
    ...  
    아이템을 생산한다.  
    ...  
    wait(empty);  
        //버퍼에 빈 공간이 생길 때까지 기다린다.  
    wait(mutex);  
        //임계 구역에 진입할 수 있을 때까지 기다린다.  
    ...  
    아이템을 버퍼에 추가한다.  
    ...  
    signal(mutex); //임계 구역을 빠져나왔다고 알려준다.  
    signal(full); //버퍼에 아이템이 있다고 알려준다.  
} while (1);
```

- 소비자 프로세스

```
do {  
    wait(full);  
        //버퍼에 아이템이 생길 때까지 기다린다.  
    wait(mutex);  
    ...  
    버퍼로부터 아이템을 가져온다.  
    ...  
    signal(mutex);  
    signal(empty);  
        //버퍼에 빈 공간이 생겼다고 알려준다.  
    ...  
    아이템을 소비한다.  
    ...  
} while (1);
```

1.14 쓰레드의 동기화(Ex1) – 생산자와 소비자 문제

- 요리사는 Table에 음식을 추가. 손님은 Table의 음식을 소비
- 요리사와 손님이 같은 객체(Table)을 공유하므로 동기화가 필요

Table

```
private ArrayList dishes
    = new ArrayList();

public void add(String dish) {
    // 테이블이 가득찼으면, 음식을 추가안함
    if(dishes.size() >= MAX_FOOD)
        return;
    dishes.add(dish);
    System.out.println("Dishes:"
        + dishes.toString());
}

public boolean remove(String dishName)
{
    // 지정된 요리와 일치하는 요리를 테이블에서 제거한다.
    for(int i=0; i<dishes.size();i++)
        if(dishName.equals(dishes.get(i)))
    {
        dishes.remove(i);
        return true;
    }
    return false;
}
```

Cook

```
public void run() {
    while(true) {
        // 임의의 요리를 하나 선택해서 table에 추가한다.
        int idx = (int)(Math.random()*table.dishNum());
        table.add(table.dishNames[idx]);
        try { Thread.sleep(1); } catch(InterruptedException e) {}
    } // while
}
```

Customer

```
public void run() {
    while(true) {
        try { Thread.sleep(10); } catch(InterruptedException e) {}
        String name = Thread.currentThread().getName();

        if(eatFood())
            System.out.println(name + " ate a " + food);
        else
            System.out.println(name + " failed to eat. :(");
    } // while
}

boolean eatFood() { return table.remove(food); }
```

main()

```
Table table = new Table(); // 여러 쓰레드가 공유하는 객체
new Thread(new Cook(table), "COOK1").start();
new Thread(new Customer(table, "donut"), "CUST1").start();
new Thread(new Customer(table, "burger"), "CUST2").start();
```

1.14 쓰레드의 동기화(Ex1) – 실행결과

[예외1] 요리사가 Table에 요리를 추가하는 과정에 손님이 요리를 먹음

[예외2] 하나 남은 요리를 손님2가 먹으려하는데, 손님1이 먹음.

【실행결과】

```
Dishes: [donut]
Dishes: [donut, burger]
Dishes: [donut, burger, donut]
Dishes: [donut, burger, donut, donut]
CUST1 ate a donut
CUST2 ate a burger
Dishes: [burger, donut, donut]
Dishes: [burger, donut, donut, burger]
Dishes: [burger, donut, donut, burger, donut]
Dishes: [burger, donut, donut, burger, donut, donut]
CUST2 ate a burger
CUST1 ate a donut
Exception in thread "COOK1" java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
    at java.util.ArrayList$Itr.next(ArrayList.java:851)
    at java.util.AbstractCollection.toString(AbstractCollection.java:461)
    at Table.add(ThreadWaitEx1.java:49)
    at Cook.run(ThreadWaitEx1.java:35)
    at java.lang.Thread.run(Thread.java:745)
CUST1 ate a donut
CUST2 ate a burger
CUST1 ate a donut
CUST2 ate a burger
CUST1 ate a donut
Exception in thread "CUST2" java.lang.IndexOutOfBoundsException: Index: 0,
Size: 0
    at java.util.ArrayList.rangeCheck(ArrayList.java:653)
    at java.util.ArrayList.get(ArrayList.java:429)
    at Table.remove(ThreadWaitEx1.java:54)
    at Customer.eatFood(ThreadWaitEx1.java:24)
    at Customer.run(ThreadWaitEx1.java:17)
    at java.lang.Thread.run(Thread.java:745)
CUST1 failed to eat. :(
CUST1 failed to eat. :(
```

1.14 쓰레드의 동기화(Ex2) – 생산자와 소비자 문제

[문제점] Table을 여러 쓰레드가 공유하기 때문에 작업 중에 끼어들기 발생

[해결책] Table의 add()와 remove()를 synchronized로 동기화

Table

```
private ArrayList dishes
    = new ArrayList();

public void add(String dish) {
    // 테이블이 가득찼으면, 음식을 추가안함
    if(dishes.size() >= MAX_FOOD)
        return;
    dishes.add(dish);
    System.out.println("Dishes:"
        + dishes.toString());
}

public boolean remove(String dishName)
{
    // 지정된 요리와 일치하는 요리를 테이블에서 제거한다.
    for(int i=0; i<dishes.size();i++)
        if(dishName.equals(dishes.get(i)))
    {
        dishes.remove(i);
        return true;
    }
    return false;
}
```



동기화된 Table

```
public synchronized void add(String dish) {
    if(dishes.size() >= MAX_FOOD)
        return;

    dishes.add(dish);
    System.out.println("Dishes:" + dishes.toString());
}

public boolean remove(String dishName) {
    synchronized(this) {
        while(dishes.size()==0) {
            String name = Thread.currentThread().getName();
            System.out.println(name+" is waiting.");
            try {
                Thread.sleep(500);
            } catch(InterruptedException e) {}
        }
        for(int i=0; i<dishes.size();i++)
            if(dishName.equals(dishes.get(i))) {
                dishes.remove(i);
                return true;
            }
    } // synchronized
    return false;
}
```

1.14 쓰레드의 동기화(Ex2) – 실행결과

[문제] 예외는 발생하지 않지만, 손님(CUST2)이 Table에 lock건 상태를 지속
요리사가 Table의 lock을 얻을 수 없어서 음식을 추가하지 못함

【실행결과】

```
Dishes: [burger]
CUST2 ate a burger
CUST1 failed to eat. :( ← donut이 없어서 먹지 못했다.
CUST2 is waiting. ← 음식이 없어서 테이블에 lock을 건 채로 계속 기다리고 있다.
CUST2 is waiting.
```

1.14 쓰레드의 동기화(Ex3) – 생산자와 소비자 문제

[문제점] 음식이 없을 때, 손님이 Table의 lock을 쥐고 안놓는다.

요리사가 lock을 얻지못해서 Table에 음식을 추가할 수 없다.

[해결책] 음식이 없을 때, wait()으로 손님이 lock을 풀고 기다리게하자.

요리사가 음식을 추가하면, notify()로 손님에게 알리자.(손님이 lock을 재획득)

```
public synchronized void add(String dish) {
    while(dishes.size() >= MAX_FOOD) {
        String name = Thread.currentThread().getName();
        System.out.println(name+" is waiting.");
        try {
            wait(); // COOK쓰레드를 기다리게 한다.
            Thread.sleep(500);
        } catch(InterruptedException e) {}
    }
    dishes.add(dish);
    notify(); // 기다리고 있는 CUST를 깨우기 위함.
    System.out.println("Dishes:" + dishes.toString());
}
```

```
public void remove(String dishName) {
    synchronized(this) {
        String name = Thread.currentThread().getName();
        while(dishes.size()==0) {
            System.out.println(name+" is waiting.");
            try {
                wait(); // CUST쓰레드를 기다리게 한다.
                Thread.sleep(500);
            } catch(InterruptedException e) {}
        }
        while(true) {
            for(int i=0; i<dishes.size();i++) {
                if(dishName.equals(dishes.get(i))) {
                    dishes.remove(i);
                    notify(); // 잠자고 있는 COOK를 깨우기 위함
                    return;
                }
            } // for문의 끝
        try {
            System.out.println(name+" is waiting.");
            wait(); // 원하는 음식이 없는 CUST쓰레드를 기다리게 한다.
            Thread.sleep(500);
        } catch(InterruptedException e) {}
    } // while(true)
} // synchronized
}
```

1.14 쓰레드의 동기화(Ex3) – 실행결과

- 전과 달리 한 쓰레드가 lock을 오래 쥐는 일이 없어짐. 효율적이 됨!!!

[실행결과]

```
Dishes:[donut]
Dishes:[donut, burger]
... 중간 생략...
Dishes:[donut, donut, donut, donut, donut, donut]
COOK1 is waiting.
CUST2 is waiting.
CUST1 ate a donut
Dishes:[donut, donut, donut, donut, donut]
CUST2 is waiting. ← 원하는 음식이 없어서 손님이 기다리고 있다.
COOK1 is waiting. ← 테이블이 가득차서 요리사가 기다리고 있다.
CUST1 ate a donut ← 테이블의 음식이 소비되어 notify()가 호출된다.
CUST2 is waiting. ← 요리사가 아닌 손님이 통지를 받고, 원하는 음식이 없어서 다시 기다린다.
CUST1 ate a donut ← 테이블의 음식이 소비되어 notify()가 호출된다.
Dishes:[donut, donut, donut, donut, donut] ← 이번엔 요리사가 통지받고 음식추가
CUST2 is waiting. ← 음식추가 통지를 받았으나 원하는 음식이 없어서 다시 기다린다.
Dishes:[donut, donut, donut, donut, donut, burger] ← 요리사가 음식추가(활동 중)
CUST1 ate a donut
CUST2 ate a burger ← 음식추가 통지를 받고, 원하는 음식을 소비(활동 중)
Dishes:[donut, donut, donut, donut, donut]
Dishes:[donut, donut, donut, donut, donut, burger]
COOK1 is waiting.
CUST1 ate a donut
```

1.15 Lock과 Condition을 이용한 동기화(1)

- java.util.concurrent.locks 패키지를 이용한 동기화(JDK1.5)

ReentrantLock 재진입이 가능한 lock. 가장 일반적인 배타 lock

ReentrantReadWriteLock 읽기에는 공유적이고, 쓰기에는 배타적인 lock

StampedLock ReentrantReadWriteLock에 낙관적인 lock의 가능을 추가

[참고] StampedLock은 JDK1.8부터 추가되었으며, 다른 lock과 달리 Lock인터페이스를 구현하지 않았다.

- 낙관적인 잠금(Optimistic Lock) : 일단 무조건 저지르고 나중에 확인

```
int getBalance() {
    long stamp = lock.tryOptimisticRead(); // 낙관적 읽기 lock을 건다.

    int curBalance = this.balance; // 공유 데이터인 balance를 읽어온다.

    if(lock.validate(stamp)) { // 쓰기 lock에 의해 낙관적 읽기 lock이 풀렸는지 확인
        stamp = lock.readLock(); // lock이 풀렸으면, 읽기 lock을 얻으려고 기다린다.

        try {
            curBalance = this.balance; // 공유 데이터를 다시 읽어온다.
        } finally {
            lock.unlockRead(stamp); // 읽기 lock을 푼다.
        }
    }

    return curBalance; // 낙관적 읽기 lock이 풀리지 않았으면 곧바로 읽어온 값을 반환
}
```

1.15 Lock과 Condition을 이용한 동기화(2)

- ReentrantLock을 이용한 동기화

```
ReentrantLock()  
ReentrantLock(boolean fair)
```

- synchronized 대신 lock()과 unlock()을 사용

```
void lock();  
void unlock();  
boolean isLocked();
```

lock을 잠근다.
lock을 해지한다.
lock이 잠겼는지 확인한다.

```
synchronized(lock) {  
    // 임계 영역  
}  
  
lock.lock();  
// 임계 영역  
lock.unlock();
```

```
lock.lock(); // ReentrantLock lock = new ReentrantLock();  
try {  
    // 임계 영역  
} finally {  
    lock.unlock();  
}
```

1.15 Lock과 Condition을 이용한 동기화(3)

- ReentrantLock과 Condition으로 쓰레드를 구분해서 wait() & notify()

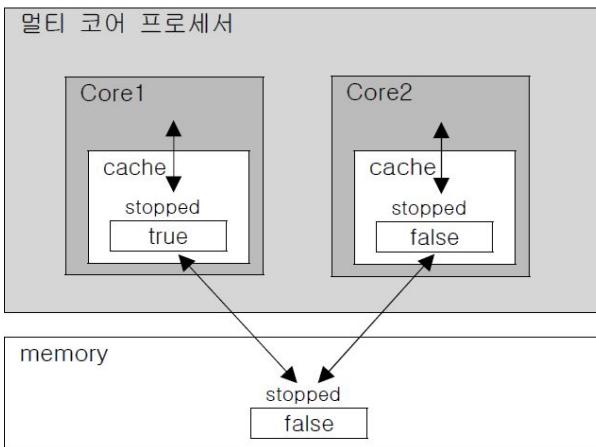
```
public void add(String dish) {  
    lock.lock();  
  
    try {  
        while(dishes.size() >= MAX_FOOD) {  
            String name = Thread.currentThread().getName();  
            System.out.println(name + " is waiting.");  
            try {  
                forCook.await(); // wait(); COOK쓰레드를 기다리게 한다.  
                Thread.sleep(500);  
            } catch(InterruptedException e) {}  
        }  
        dishes.add(dish);  
        forCust.signal(); // notify(); 기다리고 있는 CUST를 깨우기 위함.  
        System.out.println("Dishes: " + dishes.toString());  
    } finally {  
        lock.unlock();  
    }  
}
```

- ReentrantLock과 Condition의 생성방법

```
private ReentrantLock lock = new ReentrantLock(); // lock을 생성  
private Condition forCook = lock.newCondition(); // lock으로 condition을 생성  
private Condition forCust = lock.newCondition();
```

1.16 volatile – cache와 메모리간의 불일치 해소

- 성능 향상을 위해 변수의 값을 core의 cache에 저장해 놓고 작업
- 여러 쓰레드가 공유하는 변수에는 volatile을 붙여야 항상 메모리에서 읽어옴



[그림13-11] 멀티 코어 프로세서의 캐시(cache)와 메모리간의 통신

```
boolean suspended = false;  
boolean stopped = false;
```

```
volatile boolean suspended = false;  
volatile boolean stopped = false;
```

```
public void stop() {  
    stopped = true;  
}
```

```
public synchronized void stop() {  
    stopped = true;  
}
```

6.1 fork & join 프레임워크

- 작업을 여러 쓰레드가 나눠서 처리하는 것을 쉽게 해준다.(JDK1.7)
- RecursiveAction 또는 RecursiveTask를 상속받아서 구현

RecursiveAction 반환값이 없는 작업을 구현할 때 사용
RecursiveTask 반환값이 있는 작업을 구현할 때 사용

```
public abstract class RecursiveAction extends ForkJoinTask<Void> {  
    ...  
    protected abstract void compute(); // 상속을 통해 이 메서드를 구현해야 한다.  
    ...  
}
```

```
public abstract class RecursiveTask<V> {  
    ...  
    V result;  
    protected abstract V compute(); //  
    ...  
}
```

```
class SumTask extends RecursiveTask<Long> {  
    long from, to;  
  
    SumTask(long from, long to) {  
        this.from = from;  
        this.to = to;  
    }  
  
    public Long compute() {  
        // 처리할 작업을 수행하기 위한 문장을 넣는다.  
    }  
}
```

6.2 compute()의 구현

- 수행할 작업과 작업을 어떻게 나눌 것인지를 정해줘야 한다.
- `fork()`로 나눈 작업을 큐에 넣고, `compute()`를 재귀호출한다.

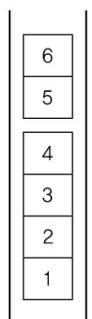
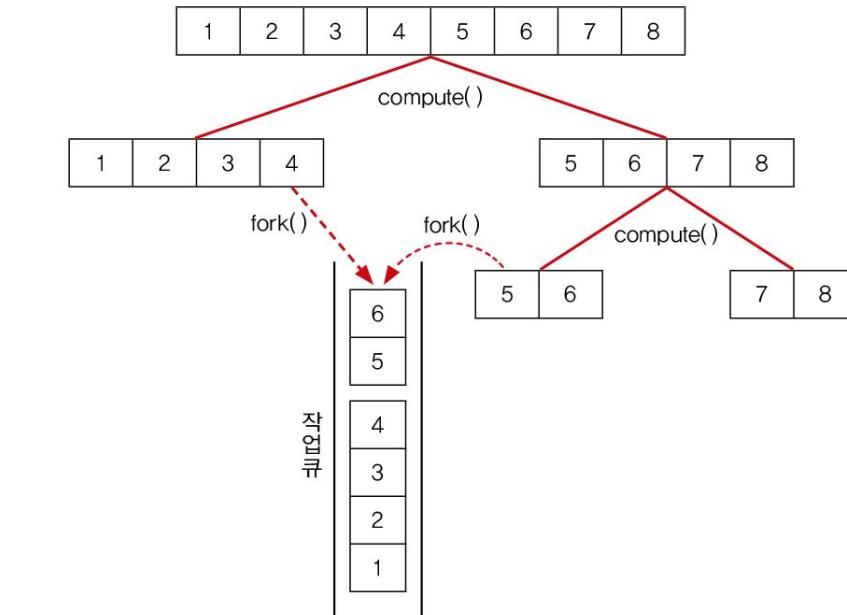
```
public Long compute() {  
    long size = to - from + 1; // from ≤ i ≤ to  
  
    if(size <= 5) // 더할 숫자가 5개 이하면  
        return sum(); // 숫자의 합을 반환.  
  
    // 범위를 반으로 나눠서 두 개의 작업을 생성  
    long half = (from+to)/2;  
  
    SumTask leftSum = new SumTask(from, half);  
    SumTask rightSum = new SumTask(half+1, to);  
  
    leftSum.fork(); // 작업(leftSum)을 작업 큐에 넣는다.  
  
    return rightSum.compute() + leftSum.join();  
}
```

```
ForkJoinPool pool = new ForkJoinPool(); // 쓰레드풀을 생성  
SumTask task = new SumTask(from, to); // 수행할 작업을 생성  
  
Long result = pool.invoke(task); // invoke()를 호출해서 작업을 시작
```

```
long sum() {  
    long tmp = 0L;  
  
    for(long i=from;i<=to;i++)  
        tmp += i;  
  
    return tmp;  
}
```

6.3 작업 훔치기(work stealing)

- 작업을 나눠서 다른 쓰레드의 작업 큐에 넣는 것



6.4 fork()와 join()

- compute()는 작업을 나누고, fork()는 작업을 큐에 넣는다.(반복)
- join()으로 작업의 결과를 합친다.(반복)

fork() 해당 작업을 쓰레드 풀의 작업 큐에 넣는다. 비동기 메서드

join() 해당 작업의 수행이 끝날 때까지 기다렸다가, 수행이 끝나면 그 결과를 반환한다. 동기 메서드

```
public Long compute() {  
    long size = to - from + 1;    // from ≤ i ≤ to  
  
    if(size <= 5)      // 더할 숫자가 5개 이하면  
        return sum(); // 숫자의 합을 반환  
  
    long half = (from+to)/2;  
  
    // 범위를 반으로 나눠서 두 개의 작업을 생성  
    SumTask leftSum = new SumTask(from, half);  
    SumTask rightSum = new SumTask(half+1, to);  
    leftSum.fork(); // 비동기 메서드. 호출 후 결과를 기다리지 않는다.  
  
    // 동기 메서드. 호출결과를 기다린다.  
    return rightSum.compute()+leftSum.join();  
}
```

람다와 스트림

(Lambda & Stream)

1.1 람다식 (Lambda Expression)이란?

- ▶ 함수(메서드)를 간단한 ‘식(Expression)’으로 표현하는 방법

```
int max(int a, int b) {  
    return a > b ? a : b;  
}
```

```
(a, b) -> a > b ? a : b
```

- ▶ 익명 함수(이름이 없는 함수, anonymous function)

```
int max(int a, int b) {  
    return a > b ? a : b;  
}
```

```
int max(int a, int b) -> {  
    return a > b ? a : b;  
}
```

- ▶ 함수와 메서드의 차이

- 근본적으로 동일. 함수는 일반적 용어, 메서드는 객체지향개념 용어
- 함수는 클래스에 독립적, 메서드는 클래스에 종속적

1.2 람다식 작성하기

- 메서드의 이름과 반환타입을 제거하고 '`->`'를 블록{} 앞에 추가한다.

```
int max(int a, int b) {  
    return a > b ? a : b;  
}
```

```
int max(int a, int b) -> {  
    return a > b ? a : b;  
}
```

- 반환값이 있는 경우, 식이나 값만 적고 `return`문 생략 가능(끝에 ';'안 붙임)

```
(int a, int b) -> {  
    return a > b ? a : b;  
}
```

```
(int a, int b) -> a > b ? a : b
```

- 매개변수의 타입이 추론 가능하면 생략 가능(대부분의 경우 생략 가능)

```
(int a, int b) -> a > b ? a : b
```

```
(a, b) -> a > b ? a : b
```

1.2 람다식 작성하기 - 주의사항

1. 매개변수가 하나인 경우, 괄호() 생략가능(타입이 없을 때만)

```
(a) -> a * a  
(int a)-> a * a
```

```
a -> a * a // OK  
int a -> a * a // 에러
```

2. 블록 안의 문장이 하나뿐 일 때, 괄호{} 생략가능(끝에 ';' 안 붙임)

```
(int i) -> {  
    System.out.println(i);  
}
```

```
(int i) -> System.out.println(i)
```

단, 하나뿐인 문장이 return문이면 괄호{} 생략불가

```
(int a, int b) -> { return a > b ? a : b; } // OK  
(int a, int b) -> return a > b ? a : b // 에러
```

1.2 람다식 작성하기 - 실습

메서드	람다식
<pre>int max(int a, int b) { return a > b ? a : b; }</pre>	①
<pre>int printVar(String name, int i) { System.out.println(name+"="+i); }</pre>	②
<pre>int square(int x) { return x * x; }</pre>	③
<pre>int roll() { return (int)(Math.random()*6); }</pre>	④

1.3 함수형 인터페이스(1/3)

- ▶ 람다식은 익명 함수? 사실은 익명 객체!!!

```
(a, b) -> a > b ? a : b
```

```
new Object() {  
    int max(int a, int b) {  
        return a > b ? a : b;  
    }  
}
```

- ▶ 람다식(익명 객체)을 다루기 위한 참조변수가 필요. 참조변수의 타입은?

```
Object obj = new Object() {  
    int max(int a, int b) {  
        return a > b ? a : b;  
    }  
};
```

타입 `obj = (a, b) -> a > b ? a : b ; // 어떤 타입?`

```
int value = obj.max(3,5); // 에러. Object클래스에 max()가 없음
```

1.3 함수형 인터페이스 (2/3)

- ▶ 함수형 인터페이스 - 단 하나의 추상 메서드만 선언된 인터페이스

```
interface MyFunction {  
    public abstract int max(int a, int b);  
}
```

```
MyFunction f = new MyFunction() {  
    public int max(int a, int b) {  
        return a > b ? a : b;  
    }  
};
```

```
int value = f.max(3,5); // OK. MyFunction에 max()가 있음
```

- ▶ 함수형 인터페이스 타입의 참조변수로 람다식을 참조할 수 있음.

(단, 함수형 인터페이스의 메서드와 람다식의 매개변수 개수와 반환타입이 일치해야 함.)

```
MyFunction f = (a, b) -> a > b ? a : b;
```

```
int value = f.max(3,5); // 실제로는 람다식(익명 함수)이 호출됨
```

1.3 함수형 인터페이스 - example

- ▶ 익명 객체를 람다식으로 대체

```
List<String> list = Arrays.asList("abc", "aaa", "bbb", "ddd", "aaa");

Collections.sort(list, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s2.compareTo(s1);
    }
});
```

```
interface Comparator<T> {
    int compare(T o1, T o2);
}
```

```
List<String> list = Arrays.asList("abc", "aaa", "bbb", "ddd", "aaa");
Collections.sort(list, (s1,s2)-> s2.compareTo(s1));
```

1.3 함수형 인터페이스 (3/3)– 매개변수와 반환타입

▶ 함수형 인터페이스 타입의 매개변수

```
void aMethod(MyFunction f) {  
    f.myMethod(); // MyFunction에 정의된 메서드 호출  
}
```

```
@FunctionalInterface  
interface MyFunction {  
    void myMethod();  
}
```

```
MyFunction f = () -> System.out.println("myMethod()");  
aMethod(f);
```

```
aMethod(() -> System.out.println("myMethod()"));
```

▶ 함수형 인터페이스 타입의 반환타입

```
MyFunction myMethod() {  
    MyFunction f = () -> {};  
    return f;  
}
```

```
MyFunction myMethod() {  
    return () -> {};  
}
```

1.4 java.util.function 패키지 (1/5)

- ▶ 자주 사용되는 다양한 함수형 인터페이스를 제공.

함수형 인터페이스	메서드	설명
java.lang. Runnable	void run()	매개변수도 없고, 반환값도 없음.
Supplier<T>	T get()	매개변수는 없고, 반환값만 있음.
Consumer<T>	T → void accept(T t)	Supplier와 반대로 매개변수만 있고, 반환값이 없음
Function<T,R>	T → R apply(T t) R →	일반적인 함수. 하나의 매개변수를 받아서 결과를 반환
Predicate<T>	T → boolean test(T t) boolean →	조건식을 표현하는데 사용됨. 매개변수는 하나, 반환 타입은 boolean

```
Predicate<String> isEmptyStr = s -> s.length()==0;  
String s = "";  
  
if(isEmptyStr.test(s)) // if(s.length()==0)  
    System.out.println("This is an empty String.");
```

1.4 java.util.function 패키지 - Quiz

Q. 아래의 빈 칸에 알맞은 함수형 인터페이스 (java.util.function 패키지)를 적으시오.

- [①] f = ()-> (int)(Math.random()*100)+1;
- [②] f = i -> System.out.print(i+", ");
- [③] f = i -> i%2==0;
- [④] f = i -> i/10*10;

1.4 java.util.function 패키지 (2/5)

▶ 매개변수가 2개인 함수형 인터페이스

함수형 인터페이스	메서드	설명
BiConsumer<T,U>	$T, U \rightarrow \boxed{\text{void accept}(T t, U u)}$	두개의 매개변수만 있고, 반환값이 없음
BiPredicate<T,U>	$T, U \rightarrow \boxed{\text{boolean test}(T t, U u)} \rightarrow \text{boolean}$	조건식을 표현하는데 사용됨. 매개변수는 둘, 반환값은 boolean
BiFunction<T,U,R>	$T, U \rightarrow \boxed{\text{R apply}(T t, U u)} \rightarrow R$	두 개의 매개변수를 받아서 하나의 결과를 반환

```
@FunctionalInterface  
interface TriFunction<T,U,V,R> {  
    R apply(T t, U u, V v);  
}
```

1.4 java.util.function 패키지 (3/5)

▶ 매개변수의 타입과 반환타입이 일치하는 함수형 인터페이스

함수형 인터페이스	메서드	설명
UnaryOperator<T>	$T \xrightarrow{\quad} T \text{ apply}(T t) \xrightarrow{\quad} T$	Function의 자손, Function과 달리 매개변수와 결과의 타입이 같다.
BinaryOperator<T>	$T, T \xrightarrow{\quad} T \text{ apply}(T t, T t) \xrightarrow{\quad} T$	BiFunction의 자손, BiFunction과 달리 매개변수와 결과의 타입이 같다.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
```

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    ...
}
```

1.4 java.util.function 패키지 (4/5)

▶ 함수형 인터페이스를 사용하는 컬렉션 프레임워크의 메서드

인터페이스	메서드	설명
Collection	boolean removeIf(Predicate<E> filter)	조건에 맞는 요소를 삭제
List	void replaceAll(UnaryOperator<E> operator)	모든 요소를 변환하여 대체
Iterable	void forEach(Consumer<T> action)	모든 요소에 작업 action을 수행
Map	V compute(K key, BiFunction<K,V,V> f)	지정된 키의 값에 작업 f를 수행
	V computeIfAbsent(K key, Function<K,V> f)	키가 없으면, 작업 f 수행 후 추가
	V computeIfPresent(K key, BiFunction<K,V,V> f)	지정된 키가 있을 때, 작업 f 수행
	V merge(K key, V value, BiFunction<V,V,V> f)	모든 요소에 병합작업 f를 수행
	void forEach(BiConsumer<K,V> action)	모든 요소에 작업 action을 수행
	void replaceAll(BiFunction<K,V,V> f)	모든 요소에 치환작업 f를 수행

```
list.forEach(i->System.out.print(i+",")); // list의 모든 요소를 출력
list.removeIf(x-> x%2==0 || x%3==0); // 2 또는 3의 배수를 제거
list.replaceAll(i->i*10); // 모든 요소에 10을 곱한다.

// map의 모든 요소를 {k,v}의 형식으로 출력
map.forEach((k,v)-> System.out.print("{ "+k+" , "+v+" } , "));
```

1.4 java.util.function 패키지 (5/5)

▶ 기본형을 사용하는 함수형 인터페이스



함수형 인터페이스	메서드	설명
DoubleToIntFunction	double → int applyAsInt(double d)	AToBFunction은 입력이 A타입 출력이 B타입
ToIntFunction<T>	T → int applyAsInt(T value)	ToBFunction은 출력이 B타입이다. 입력은 지네릭 타입
IntFunction<R>	int → R apply(int value)	AFunction은 입력이 A타입이고 출력은 지네릭 타입
ObjIntConsumer<T>	T, int → void accept(T t, int i)	ObjAFunction은 입력이 T, int 타입이고 출력은 없다.

```
Supplier<Integer> s = () -> (int) (Math.random() * 100) + 1;

static <T> void makeRandomList(Supplier<T> s, List<T> list) {
    for (int i = 0; i < 10; i++)
        list.add(s.get()); // List<Integer> list = new ArrayList<>();
```

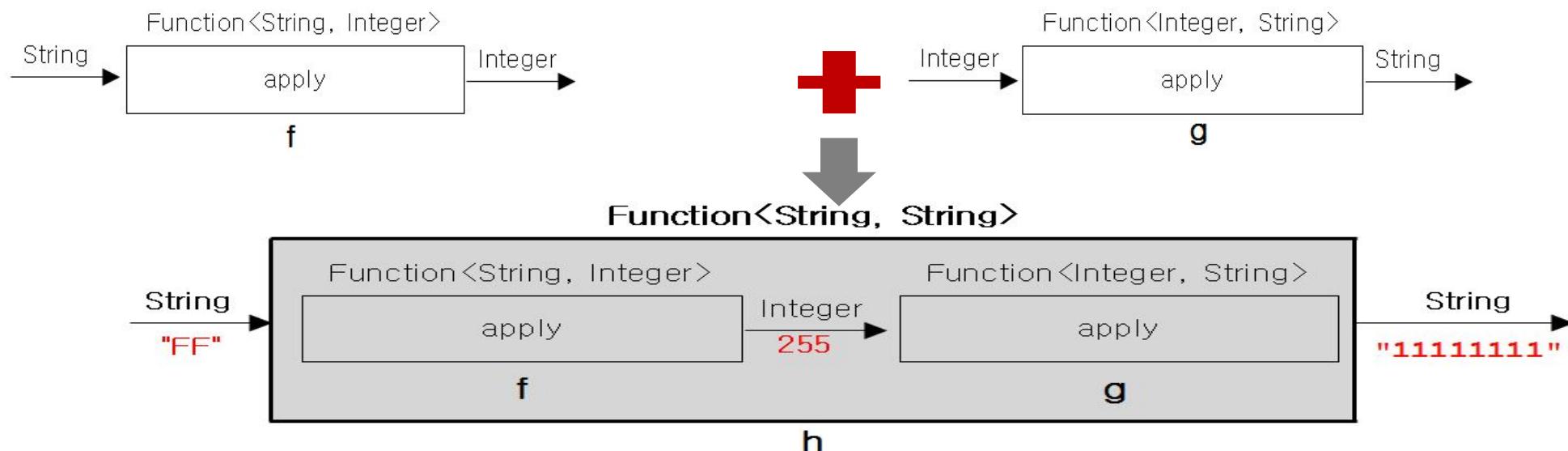
```
IntSupplier s = () -> (int) (Math.random() * 100) + 1;

static void makeRandomList(IntSupplier s, int[] arr) {
    for (int i = 0; i < arr.length; i++)
        arr[i] = s.getAsInt(); // get()이 아니라 getAsInt()임에 주의
```

1.5 Function의 합성(1/2)

▶ Function 타입의 두 람다식을 하나로 합성 – andThen()

```
Function<String, Integer> f = (s) -> Integer.parseInt(s, 16); // s를 16진 정수로 변환  
Function<Integer, String> g = (i) -> Integer.toBinaryString(i); // 2진 문자열로 변환  
Function<String, String> h = f.andThen(g); // f + g → h
```

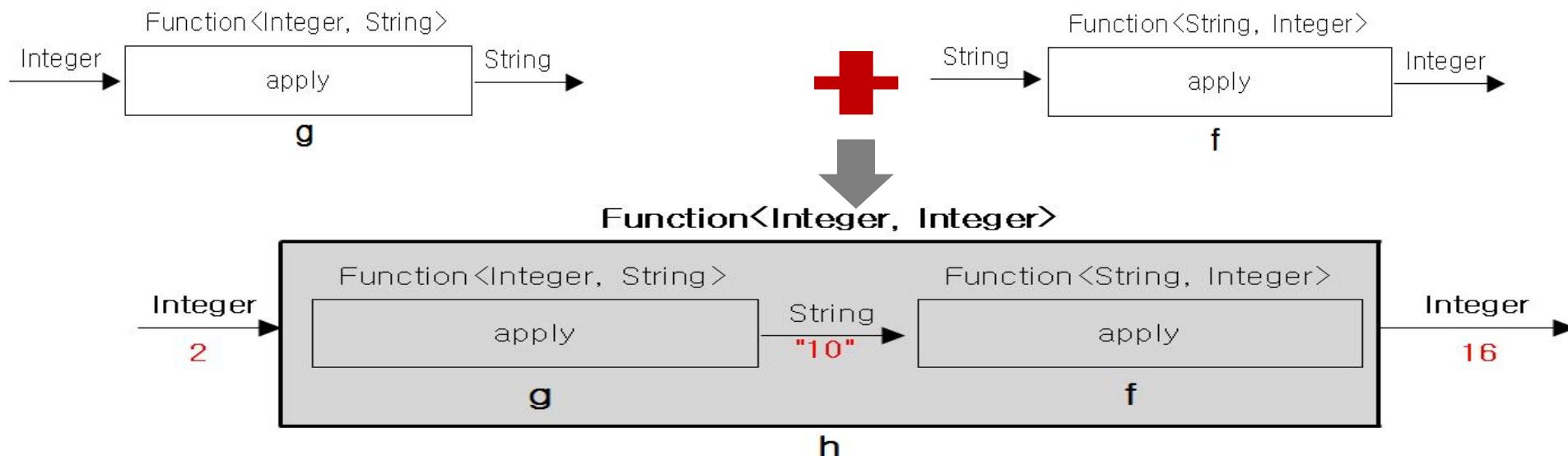


```
System.out.println(h.apply("FF")); // "FF" → 255 → "11111111"
```

1.5 Function의 합성(2/2)

▶ Function 타입의 두 람다식을 하나로 합성 – compose()

```
Function<Integer, String> g = (i) -> Integer.toBinaryString(i); // 2진 문자열로 변환  
Function<String, Integer> f = (s) -> Integer.parseInt(s, 16); // s를 16진 정수로 변환  
Function< Integer, Integer > h = f.compose(g); // g + f → h
```



```
System.out.println(h.apply(2)); // 2 → "10" → 16
```

1.6 Predicate의 결합

- ▶ **and(), or(), negate()**로 두 **Predicate**를 하나로 결합(default에서드)

```
Predicate<Integer> p = i -> i < 100;  
Predicate<Integer> q = i -> i < 200;  
Predicate<Integer> r = i -> i%2 == 0;
```

```
Predicate<Integer> notP = p.negate();          // i >= 100  
Predicate<Integer> all = notP.and(q).or(r);  // 100 <= i && i < 200 || i%2==0  
Predicate<Integer> all2 = notP.and(q.or(r)); // 100 <= i && (i < 200 || i%2==0)
```

```
System.out.println(all.test(2));    // true  
System.out.println(all2.test(2));   // false
```

- ▶ 등가비교를 위한 **Predicate**의 작성에는 **isEqual()**를 사용(static에서드)

```
Predicate<String> p = Predicate.isEqual(str1); // isEqual()은 static에서드  
Boolean result = p.test(str2); // str1과 str2가 같은지 비교한 결과를 반환
```

```
boolean result = Predicate.isEqual(str1).test(str2);
```

1.7 메서드 참조(method reference)(1/2)

- ▶ 하나의 메서드만 호출하는 람다식은 ‘메서드 참조’로 간단히 할 수 있다.

종류	람다식	메서드 참조
static메서드 참조	(x) -> ClassName.method(x)	ClassName::method
인스턴스메서드 참조	(obj, x) -> obj.method(x)	ClassName::method
특정 객체 인스턴스메서드 참조	(x) -> obj.method(x)	obj::method

- ▶ static메서드 참조

```
Integer method(String s) { // 그저 Integer.parseInt(String s)만 호출  
    return Integer.parseInt(s);  
}
```

```
int result = obj.method("123");  
int result = Integer.parseInt("123");
```

```
Function<String, Integer> f = (String s) -> Integer.parseInt(s);
```

```
Function<String, Integer> f = Integer::parseInt; // 메서드 참조
```

1.7 메서드 참조(method reference)(2/2)

▶ 인스턴스 메서드 참조

```
BiFunction<String, String, Boolean> f = (s1, s2) -> s1.equals(s2);
```



```
BiFunction<String, String, Boolean> f = String::equals;
```

▶ 특정 객체의 인스턴스 메서드 참조

```
MyClass obj = new MyClass();
Function<String, Boolean> f = (x) -> obj.equals(x); // 람다식
Function<String, Boolean> f2 = obj::equals;           // 메서드 참조
```

▶ new연산자(생성자, 배열)와 메서드 참조

```
Supplier<MyClass> s = MyClass::new;           // () -> new MyClass()
Function<Integer, MyClass> f2 = MyClass::new; // (i) -> new MyClass(i)

Function<Integer, int[]> f2 = int[]::new;      // x -> new int[x];
```

2.1 스트림(Stream)이란?

▶ 다양한 데이터 소스를 표준화된 방법으로 다루기 위한 것

```
List<Integer> list = Arrays.asList(1,2,3,4,5);
Stream<Integer> intStream = list.stream(); // 컬렉션.
Stream<String> strStream = Stream.of(new String[]{"a","b","c"}); // 배열
Stream<Integer> evenStream = Stream.iterate(0, n->n+2); // 0,2,4,6, ...
Stream<Double> randomStream = Stream.generate(Math::random); // 람다식
IntStream intStream = new Random().ints(5); // 난수 스트림 (크기가 5)
```

Stream<T> Collection.stream()

▶ 스트림이 제공하는 기능 - 중간 연산과 최종 연산

- 중간 연산 - 연산결과가 스트림인 연산. 반복적으로 적용가능
- 최종 연산 - 연산결과가 스트림이 아닌 연산. 스트림의 요소를 소모하므로 한번만 적용가능

```
stream.distinct().limit(5).sorted().forEach(System.out::println)
```

중간 연산 중간 연산 중간 연산 최종 연산

```
String[] strArr = { "dd", "aaa", "CC", "cc", "b" };
Stream<String> stream = Stream.of(strArr); // 문자열 배열이 소스인 스트림
Stream<String> filteredStream = stream.filter(); // 걸러내기 (중간 연산)
Stream<String> distinctedStream = stream.distinct(); // 중복제거 (중간 연산)
Stream<String> sortedStream = stream.sort(); // 정렬 (중간 연산)
Stream<String> limitedStream = stream.limit(5); // 스트림 자르기 (중간 연산)
int total = stream.count(); // 요소 개수 세기 (최종연산)
```

2.2 스트림(Stream)의 특징(1/2)

- ▶ 스트림은 데이터 소스로부터 데이터를 읽기만할 뿐 변경하지 않는다.

```
List<Integer> list = Arrays.asList(3,1,5,4,2);
List<Integer> sortedList = list.stream().sorted()      // list를 정렬해서
                           .collect(Collectors.toList()); // 새로운 List에 저장
System.out.println(list);           // [3, 1, 5, 4, 2]
System.out.println(sortedList);    // [1, 2, 3, 4, 5]
```

- ▶ 스트림은 Iterator처럼 일회용이다.(필요하면 다시 스트림을 생성해야 함)

```
strStream.forEach(System.out::println); // 모든 요소를 화면에 출력(최종연산)
int numOfStr = strStream.count();       // 에러. 스트림이 이미 닫혔음.
```

- ▶ 최종 연산 전까지 중간연산이 수행되지 않는다. – 지연된 연산

```
IntStream intStream = new Random().ints(1,46); // 1~45범위의 무한 스트림
intStream.distinct().limit(6).sorted()          // 중간 연산
    .forEach(i->System.out.print(i+","));     // 최종 연산
```

2.2 스트림(Stream)의 특징(2/2)

- ▶ 스트림은 작업을 내부 반복으로 처리한다.

```
for(String str : strList)
    System.out.println(str);
```

```
stream.forEach(System.out::println);
```

```
void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action); // 매개변수의 널 체크

    for(T t : src) // 내부 반복(for문을 메서드 안으로 넣음)
        action.accept(t);
}
```

- ▶ 스트림의 작업을 병렬로 처리 – 병렬스트림

```
Stream<String> strStream = Stream.of("dd", "aaa", "CC", "cc", "b");
int sum = strStream.parallelStream() // 병렬 스트림으로 전환(속성만 변경)
    .mapToInt(s -> s.length()).sum(); // 모든 문자열의 길이의 합
```

- ▶ 기본형 스트림 – IntStream, LongStream, DoubleStream

- 오토박싱 & 언박싱의 비효율이 제거됨(Stream<Integer> 대신 IntStream 사용)
- 숫자와 관련된 유용한 메서드를 Stream<T>보다 더 많이 제공

2.3 스트림의 생성(1/3)

▶ 컬렉션으로부터 스트림 생성하기

```
List<Integer> list = Arrays.asList(1,2,3,4,5);
Stream<Integer> intStream = list.stream(); // Stream<T> Collection.stream()
```

▶ 배열로부터 스트림 생성하기

```
Stream<String> strStream = Stream.of("a","b","c"); // 가변 인자
Stream<String> strStream = Stream.of(new String[]{"a","b","c"});
Stream<String> strStream = Arrays.stream(new String[]{"a","b","c"});
Stream<String> strStream = Arrays.stream(new String[]{"a","b","c"}, 0, 3);
```

▶ 특정 범위의 정수를 요소로 갖는 스트림 생성하기

```
IntStream intStream = IntStream.range(1, 5);           // 1,2,3,4
IntStream intStream = IntStream.rangeClosed(1, 5); // 1,2,3,4,5
```

2.3 스트림의 생성(2/3)

▶ 난수를 요소로 갖는 스트림 생성하기

```
IntStream intStream = new Random().ints();           // 무한 스트림  
intStream.limit(5).forEach(System.out::println); // 5개의 요소만 출력한다.
```

```
IntStream intStream = new Random().ints(5); // 크기가 5인 난수 스트림을 반환
```

```
Integer.MIN_VALUE <= ints() <= Integer.MAX_VALUE  
Long.MIN_VALUE <= longs() <= Long.MAX_VALUE  
0.0 <= doubles() < 1.0
```

* 지정된 범위의 난수를 요소로 갖는 스트림을 생성하는 메서드

```
IntStream ints(int begin, int end)                      // 무한 스트림  
LongStream longs(long begin, long end)  
DoubleStream doubles(double begin, double end)  
  
IntStream ints(long streamSize, int begin, int end)    // 유한 스트림  
LongStream longs(long streamSize, long begin, long end)  
DoubleStream doubles(long streamSize, double begin, double end)
```

2.3 스트림의 생성(3/3)

▶ 람다식을 소스로 하는 스트림 생성하기

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f) // 이전 요소에 종속적  
static <T> Stream<T> generate(Supplier<T> s)           // 이전 요소에 독립적
```

```
Stream<Integer> evenStream    = Stream.iterate(0, n->n+2); // 0,2,4,6, ...  
Stream<Double> randomStream = Stream.generate(Math::random);  
Stream<Integer> oneStream     = Stream.generate(()->1);
```

▶ 파일을 소스로 하는 스트림 생성하기

```
Stream<Path> Files.list(Path dir) // Path는 파일 또는 디렉토리
```

```
Stream<String> Files.lines(Path path)  
Stream<String> Files.lines(Path path, Charset cs)  
Stream<String> lines() // BufferedReader클래스의 메서드
```

2.4 스트림의 중간연산(1/6)

▶ 스트림 자르기 – skip(), limit()

```
Stream<T> skip(long n)          // 앞에서부터 n개 건너뛰기  
Stream<T> limit(long maxSize) // maxSize 이후의 요소는 잘라냄
```

```
IntStream skip(long n)  
IntStream limit(long maxSize)
```

```
IntStream intStream = IntStream.rangeClosed(1, 10);    // 12345678910  
intStream.skip(3).limit(5).forEach(System.out::print); // 45678
```

▶ 스트림의 요소 걸러내기 – filter(), distinct()

```
Stream<T> filter(Predicate<? super T> predicate) // 조건에 맞지 않는 요소 제거  
Stream<T> distinct()                            // 중복제거
```

```
IntStream intStream = IntStream.of(1,2,2,3,3,3,4,5,5,6);  
intStream.distinct().forEach(System.out::print);           // 123456
```

```
IntStream intStream = IntStream.rangeClosed(1, 10);        // 12345678910  
intStream.filter(i->i%2==0).forEach(System.out::print); // 246810
```

```
intStream.filter(i->i%2!=0 && i%3!=0).forEach(System.out::print);  
intStream.filter(i->i%2!=0).filter(i->i%3!=0).forEach(System.out::print);
```

2.4 스트림의 중간연산(2/6)

▶ 스트림 정렬하기 – sorted()

```
Comparator<String> CASE_INSENSITIVE_ORDER  
= new CaseInsensitiveComparator();
```

```
Stream<T> sorted() // 스트림 요소의 기본 정렬(Comparable)로 정렬  
Stream<T> sorted(Comparator<? super T> comparator) // 지정된 Comparator로 정렬
```

문자열 스트림 정렬 방법	출력결과
Stream<String> strStream = Stream.of("dd","aaa","CC","cc","b"); strStream.sorted() // 기본 정렬	
strStream.sorted(Comparator.naturalOrder()) // 기본 정렬 strStream.sorted((s1, s2) -> s1.compareTo(s2)); // 람다식도 가능 strStream.sorted(String::compareTo); // 위의 문장과 동일	CCaaabccdd
strStream.sorted(Comparator.reverseOrder()) // 기본 정렬의 역순 strStream.sorted(Comparator.<String>naturalOrder().reversed())	ddccbbaaCC
strStream.sorted(String.CASE_INSENSITIVE_ORDER) // 대소문자 구분안함	aaabCCccdd
strStream.sorted(String.CASE_INSENSITIVE_ORDER.reversed()) // 오타 아님→	ddCCccbaaa
strStream.sorted(Comparator.comparing(String::length)) // 길이 순 정렬 strStream.sorted(Comparator.comparingInt(String::length)) // no 오토박싱	bddCCccaa
strStream.sorted(Comparator.comparing(String::length).reversed())	aaaddCCccb

```
studentStream.sorted(Comparator.comparing(Student::getBan) // 반별로 정렬  
.thenComparing(Student::getTotalScore) // 총점별로 정렬  
.forEach(System.out::println);
```

2.4 스트림의 중간연산(3/6)

▶ 스트림의 요소 변환하기 – map()

```
Stream<R> map(Function<? super T,? extends R> mapper) // Stream<T>→Stream<R>
```

```
Stream<File> fileStream = Stream.of(new File("Ex1.java"), new File("Ex1")
    new File("Ex1.bak"), new File("Ex2.java"), new File("Ex1.txt"));
```

```
Stream<String> filenameStream = fileStream.map(File::getName);
filenameStream.forEach(System.out::println); // 스트림의 모든 파일의 이름을 출력
```

Stream<File>  Stream<String>

ex) 파일 스트림(Stream<File>)에서 파일 확장자(대문자)를 중복없이 뽑아내기

```
fileStream.map(File::getName)           // Stream<File> → Stream<String>
    .filter(s->s.indexOf('.')!=-1)      // 확장자가 없는 것은 제외
    .map(s->s.substring(s.indexOf('.')+1)) // Stream<String>→Stream<String>
    .map(String::toUpperCase)           // Stream<String>→Stream<String>
    .distinct() // 중복 제거
    .forEach(System.out::print); // JAVABAKTXT
```

2.4 스트림의 중간연산(4/6)

▶ 스트림을 기본 스트림으로 변환 – mapToInt(), mapToLong(), mapToDouble()

```
IntStream mapToInt(ToIntFunction<? super T> mapper) // Stream<T>→IntStream  
LongStream mapToLong(ToLongFunction<? super T> mapper) // Stream<T>→LongStream  
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper) // Stream<T>→DoubleStream
```

```
Stream<Integer> studentScoreStream = stuStream.map(Student:: getTotalScore);  
int sum = studentScoreStream.reduce(0, (a,b)-> a+b);
```



```
IntStream studentScoreStream = studentStream.mapToInt(Student:: getTotalScore);  
int allTotalScore = studentScoreStream.sum(); // IntStream의 sum()
```

int	sum()
OptionalInt	max()
OptionalInt	min()
OptionalDouble	average()

▶ 기본 스트림을 스트림으로 변환 – mapToObj(), boxed()

```
Stream<T> mapToObj(IntFunction<? extends T> mapper) // IntStream → Stream<T>  
Stream<Integer> boxed() // IntStream → Stream<Integer>
```

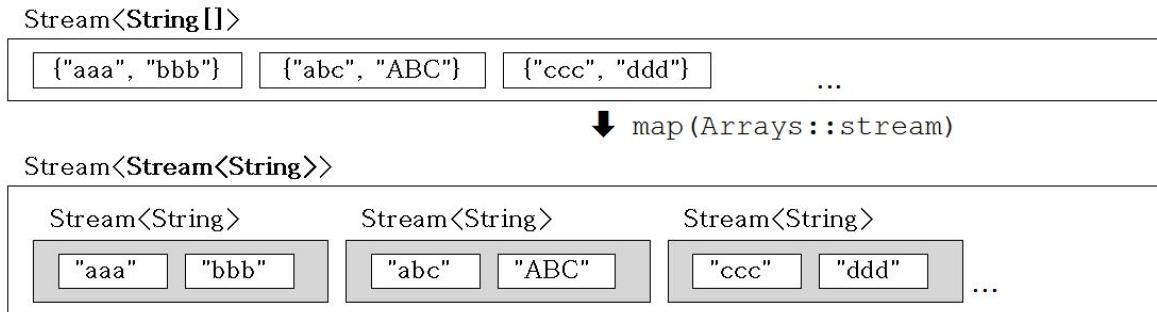
```
IntStream intStream = new Random().ints(1,46); // 1~45사이의 정수(46은 포함안됨)  
Stream<Integer> integerStream = intStream.boxed(); // IntStream → Stream<Integer>  
Stream<String> lottoStream = intStream.distinct().limit(6).sorted()  
    .mapToObj(i -> i+","); // IntStream → Stream<String>  
lottoStream.forEach(System.out::print); // 12,14,20,23,26,29,
```

2.4 스트림의 중간연산(5/6)

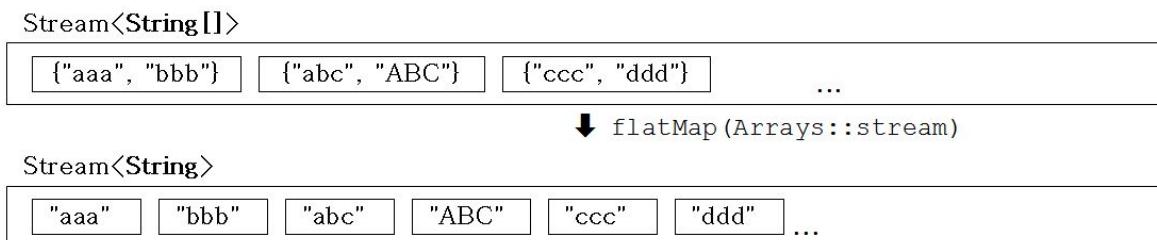
▶ 스트림의 스트림을 스트림으로 변환 – flatMap()

```
Stream<String[]> strArrStrm = Stream.of(new String[]{"abc", "def", "ghi"},  
                                         new String[]{"ABC", "GHI", "JKLMN"});
```

```
Stream<Stream<String>> strStrStrm = strArrStrm.map(Arrays::stream);
```



```
Stream<String> strStrStrm = strArrStrm.flatMap(Arrays::stream); // Arrays.stream(T[])
```



2.4 스트림의 중간연산(6/6)

▶ 스트림의 요소를 소비하지 않고 엿보기 – peek()

```
Stream<T> peek(Consumer<? super T> action) // 중간 연산(스트림을 소비X)  
void forEach(Consumer<? super T> action) // 최종 연산(스트림을 소비O)
```

```
fileStream.map(File::getName) // Stream<File> → Stream<String>  
.filter(s -> s.indexOf('.')!=-1) // 확장자가 없는 것은 제외  
.peek(s->System.out.printf("filename=%s%n", s)) // 파일명을 출력한다.  
.map(s -> s.substring(s.indexOf('.')+1)) // 확장자만 추출  
.peek(s->System.out.printf("extension=%s%n", s)) // 확장자를 출력한다.  
.forEach(System.out::println); // 최종연산 스트림을 소비.
```

2.5 Optional<T>과 OptionallInt(1/2)

▶ 'T'타입 객체의 래퍼클래스 – Optional<T>

```
String str = "abc";
Optional<String> optVal = Optional.of(str);
Optional<String> optVal = Optional.of("abc");
Optional<String> optVal = Optional.of(null);           // NullPointerException 발생
Optional<String> optVal = Optional.ofNullable(null); // OK
```

```
public final class Optional<T> {
    private final T value
    ...
}
```

▶ Optional객체의 값 가져오기 – get(), orElse(), orElseGet(), orElseThrow()

```
Optional<String> optVal = Optional.of("abc");
String str1 = optVal.get();                      // optVal에 저장된 값을 반환. null이면 예외발생
String str2 = optVal.orElse("");                // optVal에 저장된 값이 null일 때는, ""를 반환
String str3 = optVal.orElseGet(String::new); // 람다식 사용가능 () -> new String()
String str4 = optVal.orElseThrow(NullPointerException::new); // 널이면 예외발생
```

T orElseGet(Supplier<? extends T> other)

T orElseThrow(Supplier<? extends X> exceptionSupplier)

▶ isPresent() – Optional객체의 값이 null이면 false, 아니면 true를 반환

```
if(Optional.ofNullable(str).isPresent()) { // if(str!=null) {
    System.out.println(str);
}
```

// ifPresent(Consumer) - 널이 아닐때만 작업 수행, 널이면 아무 일도 안 함
Optional.ofNullable(str).ifPresent(System.out::println);

2.5 Optional<T>과 OptionallInt(2/2)

- ▶ 기본형 값을 감싸는 래퍼클래스 – OptionallInt, OptionalLong, OptionalDouble

```
public final class OptionallInt {  
    ...  
    private final boolean isPresent; // 값이 저장되어 있으면 true  
    private final int value; // int타입의 변수
```

- ▶ OptionallInt의 값 가져오기 – int getAsInt()

Optional클래스	값을 반환하는 메서드	
Optional<T>	T	get()
OptionallInt	int	getAsInt()
OptionalLong	long	getAsLong()
OptionalDouble	double	getAsDouble()

- ▶ 빈 Optional객체의 비교

```
OptionallInt opt1 = OptionallInt.of(0); // OptionallInt에 0을 저장  
OptionallInt opt2 = OptionallInt.empty(); // 빈 OptionallInt객체. OptionallInt에 0이 저장됨  
Optional<String> opt3 = Optional.ofNullable(null); // null이 저장된 Optionall 객체  
Optional<String> opt4 = Optional.empty(); // 빈 Optionall 객체. null이 저장됨  
System.out.println(opt1.equals(opt2)); // false  
System.out.println(opt3.equals(opt4)); // true
```

2.6 스트림의 최종연산(1/4)

- ▶ 스트림의 모든 요소에 지정된 작업을 수행 – `forEach()`, `forEachOrdered()`

```
void forEach(Consumer<? super T> action)           // 병렬스트림인 경우 순서가 보장되지 않음  
void forEachOrdered(Consumer<? super T> action) // 병렬스트림인 경우에도 순서가 보장됨
```

```
IntStream.range(1, 10).sequential().forEach(System.out::print);      // 123456789  
IntStream.range(1, 10).sequential().forEachOrdered(System.out::print); // 123456789
```

```
IntStream.range(1, 10).parallel().forEach(System.out::print);        // 683295714  
IntStream.range(1, 10).parallel().forEachOrdered(System.out::print); // 123456789
```

- ▶ 스트림을 배열로 변환 – `toArray()`

```
Object[] toArray()                                // 스트림의 모든 요소를 Object배열에 담아 반환  
A[]      toArray(IntFunction<A[]> generator) // 스트림의 모든 요소를 A타입의 배열에 담아 반환
```

```
Student[] stuNames = studentStream.toArray(Student[]::new); // OK. x-> new Student[x]  
Student[] stuNames = studentStream.toArray(); // 에러.  
Object[]  stuNames = studentStream.toArray(); // OK.
```

2.6 스트림의 최종연산(2/4)

▶ 조건 검사 – allMatch(), anyMatch(), noneMatch()

```
boolean allMatch (Predicate<? super T> predicate) // 모든 요소가 조건을 만족시키면 true  
boolean anyMatch (Predicate<? super T> predicate) // 한 요소라도 조건을 만족시키면 true  
boolean noneMatch(Predicate<? super T> predicate) // 모든 요소가 조건을 만족시키지 않으면 true
```

```
boolean hasFailedStu = stuStream.anyMatch(s-> s.getTotalScore()<=100); // 낙제자가 있는지?
```

▶ 조건에 일치하는 요소 찾기 – findFirst() , findAny()

```
Optional<T> findFirst() // 첫 번째 요소를 반환. 순차 스트림에 사용  
Optional<T> findAny() // 아무거나 하나를 반환. 병렬 스트림에 사용
```

```
Optional<Student> result = stuStream.filter(s-> s.getTotalScore() <= 100).findFirst();  
Optional<Student> result = parallelStream.filter(s-> s.getTotalScore() <= 100).findAny();
```

2.6 스트림의 최종연산(3/4)

- ▶ 스트림에 대한 통계정보 제공 – `count()`, `sum()`, `average()`, `max()`, `min()`

```
long      count()
Optional<T> max(Comparator<? super T> comparator)
Optional<T> min(Comparator<? super T> comparator)
```

Stream<T>

```
long      count()
Int       sum()
OptionalDouble average()
OptionalInt   max()
OptionalInt   min()
IntSummaryStatistics summaryStatistics()
```

IntStream

```
double getAverage()
long   getCount()
int    getMax()
int    getMin()
long   getSum()
```

IntSummaryStatistics

2.6 스트림의 최종연산(4/4)

▶ 스트림의 요소를 하나씩 줄여가며 누적연산 수행 – reduce()

```
Optional<T> reduce(BinaryOperator<T> accumulator)
T          reduce(T identity, BinaryOperator<T> accumulator)
U          reduce(U identity, BiFunction<U,T,U> accumulator, BinaryOperator<U> combiner)
```

- identity - 초기값
- accumulator - 이전 연산결과와 스트림의 요소에 수행할 연산
- combiner - 병렬처리된 결과를 합치는데 사용할 연산(병렬 스트림)

```
// int reduce(int identity, IntBinaryOperator op)
int count = intStream.reduce(0, (a,b) -> a + 1);                                // count()
int sum   = intStream.reduce(0, (a,b) -> a + b);                                // sum()
int max   = intStream.reduce(Integer.MIN_VALUE, (a,b)-> a > b ? a : b); // max()
int min   = intStream.reduce(Integer.MAX_VALUE, (a,b)-> a < b ? a : b); // min()
```

```
int a = identity;
for(int b : stream)
    a = a + b; // sum()
```

```
// OptionalInt reduce(IntBinaryOperator accumulator)
OptionalInt max = intStream.reduce((a,b) -> a > b ? a : b); // max()
OptionalInt min = intStream.reduce((a,b) -> a < b ? a : b); // min()
```

```
OptionalInt max = intStream.reduce(Integer::max); // static int max(int a, int b)
OptionalInt min = intStream.reduce(Integer::min); // static int min(int a, int b)
```

2.7 collect(), Collector, Collectors

▶ collect()는 Collector를 매개변수로 하는 스트림의 최종연산

```
Object collect(Collector collector) // Collector를 구현한 클래스의 객체를 매개변수로  
Object collect(Supplier supplier, BiConsumer accumulator, BiConsumer combiner) // 잘 안쓰임
```

▶ Collector는 수집(collect)에 필요한 메서드를 정의해 놓은 인터페이스

```
public interface Collector<T, A, R> { // T(요소)를 A에 누적한 다음, 결과를 R로 변환해서 반환  
    Supplier<A>           supplier();          // StringBuilder::new           누적할 곳  
    BiConsumer<A, T>      accumulator();       // (sb, s) -> sb.append(s)   누적방법  
    BinaryOperator<A>     combiner();         // (sb1, sb2) -> sb1.append(sb2) 결합방법(병렬)  
    Function<A, R>        finisher();         // sb -> sb.toString()      최종변환  
    Set<Characteristics> characteristics(); // 컬렉터의 특성이 담긴 Set을 반환  
    ...  
}
```

▶ Collectors클래스는 다양한 기능의 컬렉터(Collector를 구현한 클래스)를 제공

- 변환 - mapping(), toList(), toSet(), toMap(), toCollection(), ...
- 통계 - counting(), summingInt(), averagingInt(), maxBy(), minBy(), summarizingInt(), ...
- 문자열 결합 - joining()
- 리듀싱 - reducing()
- 그룹화와 분할 - groupingBy(), partitioningBy(), collectingAndThen()

2.8 Collectors의 메서드(1/4)

▶ 스트림을 컬렉션으로 변환 – `toList()`, `toSet()`, `toMap()`, `toCollection()`

```
List<String> names = stuStream.map(Student::getName) // Stream<Student>->Stream<String>
    .collect(Collectors.toList()); // Stream<String>->List<String>
ArrayList<String> list = names.stream()
    .collect(Collectors.toCollection(ArrayList::new)); // Stream<String>->ArrayList<String>

Map<String, Person> map = personStream
    .collect(Collectors.toMap(p->p.getRegId(), p->p)); // Stream<Person>->Map<String, Person>
```

▶ 스트림의 통계정보 제공 – `counting()`, `summingInt()`, `maxBy()`, `minBy()`, ...

```
long count = stuStream.count();
long count = stuStream.collect(Collectors.counting()); // Collectors.counting()
```

```
long totalScore = stuStream.mapToInt(Student::getTotalScore).sum(); // IntStream의 sum()
long totalScore = stuStream.collect(Collectors.summingInt(Student::getTotalScore));
```

```
OptionalInt topScore = studentStream.mapToInt(Student::getTotalScore).max();
Optional<Student> topStudent = stuStream
    .max(Comparator.comparingInt(Student::getTotalScore));
Optional<Student> topStudent = stuStream
    .collect(Collectors.maxBy(Comparator.comparingInt(Student::getTotalScore)));
```

2.8 Collectors의 메서드(2/4)

▶ 스트림을 리듀싱 – reducing()

```
Collector<T> reducing(BinaryOperator<T> op)
Collector<T> reducing(T identity, BinaryOperator<T> op)
Collector<U> reducing(U identity, Function<T,U> mapper, BinaryOperator<U> op) // map+reduce
```

```
IntStream intStream = new Random().ints(1,46).distinct().limit(6);

OptionalInt max = intStream.reduce(Integer::max);
Optional<Integer> max = intStream.boxed().collect(reducing(Integer::max));
```

```
long sum = intStream.reduce(0, (a,b) -> a + b);
long sum = intStream.boxed().collect(reducing(0, (a,b)-> a + b));
```

```
int grandTotal = stuStream.map(Student::getTotalScore).reduce(0, Integer::sum);
int grandTotal = stuStream.collect(reducing(0, Student::getTotalScore, Integer::sum));
```

▶ 문자열 스트림의 요소를 모두 연결 – joining()

```
String studentNames = stuStream.map(Student::getName).collect(joining());
String studentNames = stuStream.map(Student::getName).collect(joining(", ")); // 구분자
String studentNames = stuStream.map(Student::getName).collect(joining(", ", "[", "]"));
String studentInfo = stuStream.collect(joining(", ")); // Student의 toString()으로 결합
```

2.8 Collectors의 메서드(3/4)

▶ 스트림의 요소를 2분할 – partitioningBy()

```
Collector<T, R, M> partitioningBy(Predicate predicate)
Collector<T, R, M> partitioningBy(Predicate predicate, Collector downstream)
```

```
Map<Boolean, List<Student>> stuBySex = stuStream
    .collect(partitioningBy(Student::isMale)); // 학생들을 성별로 분할
List<Student> maleStudent = stuBySex.get(true); // Map에서 남학생 목록을 얻는다.
List<Student> femaleStudent = stuBySex.get(false); // Map에서 여학생 목록을 얻는다.
```

```
Map<Boolean, Long> stuNumBySex = stuStream
    .collect(partitioningBy(Student::isMale, counting())); // 분할 + 통계
System.out.println("남학생 수 :" + stuNumBySex.get(true)); // 남학생 수 :8
System.out.println("여학생 수 :" + stuNumBySex.get(false)); // 여학생 수 :10
```

```
Map<Boolean, Optional<Student>> topScoreBySex = stuStream // 분할 + 통계
    .collect(partitioningBy(Student::isMale, maxBy(comparingInt(Student::getScore))));
```

```
System.out.println("남학생 1등 :" + topScoreBySex.get(true)); // 남학생 1등 :Optional[[나자바, 남, 1, 1,300]]
System.out.println("여학생 1등 :" + topScoreBySex.get(false)); // 여학생 1등 :Optional[[김지미, 여, 1, 1,250]]
```

```
Map<Boolean, Map<Boolean, List<Student>>> failedStuBySex = stuStream // 다중 분할
    .collect(partitioningBy(Student::isMale, // 1. 성별로 분할(남/녀)
        partitioningBy(s -> s.getScore() < 150))); // 2. 성적으로 분할(불합격/합격)
List<Student> failedMaleStu = failedStuBySex.get(true).get(true);
List<Student> failedFemaleStu = failedStuBySex.get(false).get(true);
```

2.8 Collectors의 메서드(4/4)

▶ 스트림의 요소를 그룹화 – groupingBy()

```
Collector groupingBy(Function classifier)
Collector groupingBy(Function classifier, Collector downstream)
Collector groupingBy(Function classifier, Supplier mapFactory, Collector downstream)
```

```
Map<Integer, List<Student>> stuByBan = stuStream           // 학생을 반별로 그룹화
    .collect(groupingBy(Student::getBan, toList())); // toList() 생략 가능
```

```
Map<Integer, Map<Integer, List<Student>>> stuByHakAndBan = stuStream // 다중 그룹화
    .collect(groupingBy(Student::getHak,                      // 1. 학년별 그룹화
                        groupingBy(Student::getBan))           // 2. 반별 그룹화
    );
```

```
Map<Integer, Map<Integer, Set<Student.Level>>> stuByHakAndBan = stuStream
.collect(
    groupingBy(Student::getHak, groupingBy(Student::getBan,   // 다중 그룹화(학년별, 반별)
        mapping(s-> { // 성적등급(Level)으로 변환. List<Student> → Set<Student.Level>
            if      (s.getScore() >= 200) return Student.Level.HIGH;
            else if(s.getScore() >= 100) return Student.Level.MID;
            else                           return Student.Level.LOW;
        } , toSet()) // mapping()                                // enum Level { HIGH, MID, LOW }
    )) // groupingBy()
); // collect()
```

2.9 Collector 구현하기

▶ Collector 인터페이스를 구현하는 클래스를 작성

```
public interface Collector<T, A, R> { // T(요소)를 A에 누적한 다음, 결과를 R로 변환해서 반환
    Supplier<A> supplier(); // 결과를 저장할 공간(A)을 제공
    BiConsumer<A, T> accumulator(); // 스트림의 요소(T)를 수집(collect) 할 방법을 제공
    BinaryOperator<A> combiner(); // 두 저장공간(A)을 병합할 방법을 제공(병렬 스트림)
    Function<A, R> finisher(); // 최종변환(A → R). 변환할 필요가 없는 경우, x->x
    Set<Characteristics> characteristics(); // 컬렉터의 특성이 담긴 Set을 반환
    ...
}
```

▶ 컬렉터가 수행할 작업의 속성 정보를 제공 – characteristics()

Characteristics.CONCURRENT	병렬로 처리할 수 있는 작업
Characteristics.UNORDERED	스트림의 요소의 순서가 유지될 필요가 없는 작업
Characteristics.IDENTITY_FINISH	finisher()가 항등 함수인 작업

```
public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(EnumSet.of(
        Collector.Characteristics.CONCURRENT, Collector.Characteristics.UNORDERED
    ));
}
```

```
Set<Characteristics> characteristics() {
    return Collections.emptySet(); // 지정할 특성이 없으면 빈 set을 반환
}
```

2.9 Collector 구현하기 – example

▶ 문자열 스트림의 모든 요소를 연결하는 컬렉터 - ConcatCollector

```
class ConcatCollector implements Collector<String, StringBuilder, String> {
    public Supplier<StringBuilder> supplier() {
        return () -> new StringBuilder(); // return StringBuilder::new;
    }

    public BiConsumer<StringBuilder, String> accumulator() {
        return (sb,s) -> sb.append(s);
    }

    public Function<StringBuilder, String> finisher() {
        return sb -> sb.toString();
    }

    public BinaryOperator<StringBuilder> combiner() {
        return (sb, sb2) -> sb.append(sb2);
    }

    public Set<Characteristics> characteristics() {
        return Collections.emptySet();
    }
}
```

```
String[] strArr = {"aaa","bbb","ccc" };
// supplier()
StringBuffer sb = new StringBuffer();

for(String tmp : strArr)
    sb.append(tmp); // accumulator()
// finisher()
String result = sb.toString();
```

```
public static void main(String[] args) {
    String[] strArr = { "aaa","bbb","ccc" };
    Stream<String> strStream = Stream.of(strArr);
    String result = strStream.collect(new ConcatCollector());
    System.out.println("result="+result); // result=aaabbbccc
}
```

2.10 스트림의 변환(1/2)

from	to	변환 메서드
1. 스트림 → 기본형 스트림		
Stream<T>	IntStream LongStream DoubleStream	mapToInt(ToIntFunction<T> mapper) mapToLong(ToLongFunction<T> mapper) mapToDouble(ToDoubleFunction<T> mapper)
2. 기본형 스트림 → 스트림		
IntStream LongStream DoubleStream	Stream<Integer> Stream<Long> Stream<Double> Stream<U>	boxed() mapToObj(DoubleFunction<U> mapper)
3. 기본형 스트림 → 기본형 스트림		
IntStream LongStream DoubleStream	LongStream DoubleStream	asLongStream() asDoubleStream()
4. 스트림 → 부분 스트림		
Stream<T> IntStream	Stream<T> IntStream	skip(long n) limit(long maxSize)
5. 두 개의 스트림 → 스트림		
Stream<T>, Stream<T> IntStream, IntStream LongStream, LongStream DoubleStream, DoubleStream	Stream<T> IntStream LongStream DoubleStream	concat(Stream<T> a, Stream<T> b) concat(IntStream a, IntStream b) concat(LongStream a, LongStream b) concat(DoubleStream a, DoubleStream b)
6. 스트림의 스트림 → 스트림		
Stream<Stream<T>> Stream<IntStream> Stream<LongStream> Stream<DoubleStream>	Stream<T> IntStream LongStream DoubleStream	flatMap(Function mapper) flatMapToInt(Function mapper) flatMapToLong(Function mapper) flatMaptoDouble(Function mapper)

2.10 스트림의 변환(2/2)

from	to	변환 메서드
7. 스트림 ↔ 병렬 스트림		
Stream<T>	Stream<T>	
IntStream	IntStream	parallel() // 스트림 → 병렬 스트림
LongStream	LongStream	sequential() // 병렬 스트림 → 스트림
DoubleStream	DoubleStream	
8. 스트림 → 컬렉션		
Stream<T>	Collection<T>	collect(Collectors.toCollection(Supplier factory))
IntStream	List<T>	collect(Collectors.toList())
LongStream		
DoubleStream	Set<T>	collect(Collectors.toSet())
9. 컬렉션 → 스트림		
Collection<T>		
List<T>	Stream<T>	stream()
Set<T>		
10. 스트림 → Map		
Stream<T>		collect(Collectors.toMap(Function key, Function value))
IntStream	Map<K,V>	collect(Collectors.toMap(Function, Function, BinaryOperator))
LongStream		collect(Collectors.toMap(Function, Function, BinaryOperator merge, Supplier mapSupplier))
DoubleStream		
11. 스트림 → 배열		
Stream<T>	Object[]	toArray()
	T []	toArray(IntFunction<A []> generator)
IntStream	int []	
LongStream	long []	
DoubleStream	double []	toArray()

입출력(I/O)

1. 입출력(I/O)

1.1 입출력(I/O)과 스트림(stream)

1.2 바이트기반 스트림 – InputStream, OutputStream

1.3 보조스트림

1.4 문자기반 스트림 – Reader, Writer

2. 바이트기반 스트림

2.1 InputStream과 OutputStream

2.2 ByteArrayInputStream과 ByteArrayOutputStream

2.3 FileInputStream과 FileOutputStream

3. 바이트기반 보조스트림

3.1 FilterInputStream과 FilterOutputStream

3.2 BufferedInputStream과 BufferedOutputStream

3.3 DataInputStream과 DataOutputStream

3.4 SequenceInputStream

3.5 PrintStream

4. 문자기반 스트림

4.1 Reader와 Writer

4.2 FileReader와 FileWriter

4.3 PipedReader와 PipedWriter

4.4 StringReader와 StringWriter

5. 문자기반 보조스트림

5.1 BufferedReader와 BufferedWriter

5.2 InputStreamReader와 OutputStreamWriter

6. 표준입출력과 File

6.1 표준입출력

6.2 RandomAccessFile

6.3 File

7. 직렬화(serialization)

7.1 직렬화(serialization)란?

7.2 ObjectInputStream, ObjectOutputStream

7.3 직렬화 가능한 클래스 만들기

7.4 직렬화 가능한 클래스의 버전관리

1. 입출력(I/O)

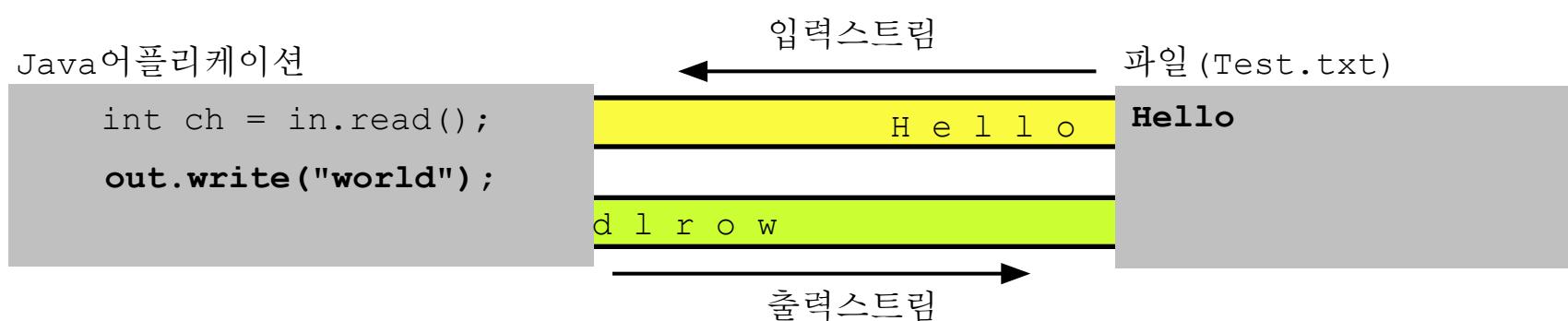
1.1 입출력(I/O)과 스트림(stream)

▶ 입출력(I/O)이란?

- 입력(Input)과 출력(Output)을 줄여 부르는 말
- 두 대상 간의 데이터를 주고 받는 것

▶ 스트림(stream)이란?

- 데이터를 운반(입출력)하는데 사용되는 연결통로
- 연속적인 데이터의 흐름을 물(stream)에 비유해서 붙여진 이름
- 하나의 스트림으로 입출력을 동시에 수행할 수 없다.(단방향 통신)
- 입출력을 동시에 수행하려면, 2개의 스트림이 필요하다.



1.2 바이트기반 스트림 – InputStream, OutputStream

- 데이터를 바이트(byte)단위로 주고 받는다.

InputStream	OutputStream
abstract int read()	abstract void write(int b)
int read(byte[] b)	void write(byte[] b)
int read(byte[] b, int off, int len)	void write(byte[] b, int off, int len)

```
public abstract class InputStream {  
    ...  
    // 입력스트림으로부터 1 byte를 읽어서 반환한다. 읽을 수 없으면 -1을 반환한다.  
    abstract int read();  
  
    // 입력스트림으로부터 len개의 byte를 읽어서 byte배열 b의 off위치부터 저장한다.  
    int read(byte[] b, int off, int len) {  
        ...  
        for(int i=off; i < off + len; i++) {  
            // read()를 호출해서 데이터를 읽어서 배열을 채운다.  
            b[i] = (byte)read();  
        }  
        ...  
    }  
    // 입력스트림으로부터 byte배열 b의 크기만큼 데이터를 읽어서 배열 b에 저장한다.  
    int read(byte[] b) {  
        return read(b, 0, b.length);  
    }  
    ...  
}
```

입력스트림	출력스트림	대상
FileInputStream	FileOutputStream	파일
ByteArrayInputStream	ByteArrayOutputStream	메모리
PipedInputStream	PipedOutputStream	프로세스
AudioInputStream	AudioOutputStream	오디오장치

1.3 보조스트림

- 스트림의 기능을 향상시키거나 새로운 기능을 추가하기 위해 사용
- 독립적으로 입출력을 수행할 수 없다.

```
// 먼저 기반스트림을 생성한다.  
FileInputStream fis = new FileInputStream("test.txt");  
// 기반스트림을 이용해서 보조스트림을 생성한다.  
BufferedInputStream bis = new BufferedInputStream(fis);  
  
bis.read(); // 보조스트림인 BufferedInputStream으로부터 데이터를 읽는다.
```

입력	출력	설명
FilterInputStream	FilterOutputStream	필터를 이용한 입출력 처리
BufferedInputStream	BufferedOutputStream	버퍼를 이용한 입출력 성능향상
DataInputStream	DataOutputStream	int, float와 같은 기본형 단위(primitive type)로 데이터를 처리하는 기능
SequenceInputStream	SequenceOutputStream	두 개의 스트림을 하나로 연결
LineNumberInputStream	없음	읽어 온 데이터의 라인 번호를 카운트 (JDK1.1부터 LineNumberReader로 대체)
ObjectInputStream	ObjectOutputStream	데이터를 객체단위로 읽고 쓰는데 사용. 주로 파일을 이용하여 객체 직렬화와 관련있음
없음	PrintStream	버퍼를 이용하여, 추가적인 print관련 기능(print, printf, println메서드)
PushbackInputStream	없음	버퍼를 이용해서 읽어 온 데이터를 다시 되돌리는 기능 (unread, push back to buffer)

1.4 문자기반 스트림 – Reader, Writer

- 입출력 단위가 문자(char, 2 byte)인 스트림. 문자기반 스트림의 최고조상

바이트기반 스트림	문자기반 스트림	대상
<code>FileInputStream</code> <code> FileOutputStream</code>	<code>FileReader</code> <code> FileWriter</code>	파일
<code>ByteArrayInputStream</code> <code>ByteArrayOutputStream</code>	<code>CharArrayReader</code> <code>CharArrayWriter</code>	메모리
<code>PipedInputStream</code> <code>PipedOutputStream</code>	<code>PipedReader</code> <code>PipedWriter</code>	프로세스
<code>StringBufferInputStream</code> <code>StringBufferOutputStream</code>	<code>StringReader</code> <code>StringWriter</code>	메모리

바이트기반 보조스트림	문자기반 보조스트림
<code>BufferedInputStream</code> <code>BufferedOutputStream</code>	<code>BufferedReader</code> <code>BufferedWriter</code>
<code>FilterInputStream</code> <code>FilterOutputStream</code>	<code>FilterReader</code> <code>FilterWriter</code>
<code>LineNumberInputStream</code>	<code>LineNumberReader</code>
<code>PrintStream</code>	<code>PrintWriter</code>
<code>PushbackInputStream</code>	<code>PushbackReader</code>

`InputStream` → Reader

`OutputStream` → Writer

<code>InputStream</code>	Reader
<code>abstract int read()</code> <code>int read(byte[] b)</code> <code>int read(byte[] b, int off, int len)</code>	<code>int read()</code> <code>int read(char[] cbuf)</code> <code>abstract int read(char[] cbuf, int off, int len)</code>
<code>OutputStream</code>	Writer
<code>abstract void write(int b)</code> <code>void write(byte[] b)</code> <code>void write(byte[] b, int off, int len)</code>	<code>void write(int c)</code> <code>void write(char[] cbuf)</code> <code>abstract void write(char[] cbuf, int off, int len)</code> <code>void write(String str)</code> <code>void write(String str, int off, int len)</code>

2. 바이트기반 스트림

2.1 InputStream과 OutputStream

▶ InputStream(바이트기반 입력스트림의 최고 조상)의 메서드

메서드명	설 명
int available()	스트림으로부터 읽어 올 수 있는 데이터의 크기를 반환한다.
void close()	스트림을 닫음으로써 사용하고 있던 자원을 반환한다.
void mark(int readlimit)	현재위치를 표시해 놓는다. 후에 reset()에 의해서 표시해 놓은 위치로 다시 돌아갈 수 있다. readlimit은 되돌아갈 수 있는 byte의 수이다.
boolean markSupported()	mark()와 reset()을 지원하는지를 알려 준다. mark()와 reset() 기능을 지원하는 것은 선택적이므로, mark()와 reset()을 사용하기 전에 markSupported()를 호출해서 지원 여부를 확인해야 한다.
abstract int read()	1 byte를 읽어 온다(0~255사이의 값). 더 이상 읽어 올 데이터가 없으면 -1을 반환한다. abstract메서드라서 InputStream의 자손들은 자신의 상황에 알맞게 구현해야 한다.
int read(byte[] b)	배열 b의 크기만큼 읽어서 배열을 채우고 읽어 온 데이터의 수를 반환한다. 반환하는 값은 항상 배열의 크기보다 작거나 같다.
int read(byte[] b, int off, int len)	최대 len개의 byte를 읽어서, 배열 b의 지정된 위치(off)부터 저장한다. 실제로 읽어 올 수 있는 데이터가 len개보다 적을 수 있다.
void reset()	스트림에서의 위치를 마지막으로 mark()이 호출되었던 위치로 되돌린다.
long skip(long n)	스트림에서 주어진 길이(n)만큼을 건너뛴다.

▶ OutputStream(바이트기반 출력스트림의 최고 조상)의 메서드

메서드명	설 명
void close()	입력소스를 닫음으로써 사용하고 있던 자원을 반환한다.
void flush()	스트림의 버퍼에 있는 모든 내용을 출력소스에 쓴다.
abstract void write(int b)	주어진 값을 출력소스에 쓴다.
void write(byte[] b)	주어진 배열 b에 저장된 모든 내용을 출력소스에 쓴다.
void write(byte[] b, int off, int len)	주어진 배열 b에 저장된 내용 중에서 off번째부터 len개 만큼만 읽어서 출력소스에 쓴다.

2.2 ByteArrayInputStream과 ByteArrayOutputStream

- 바이트배열(byte[])에 데이터를 입출력하는 바이트기반 스트림

```
import java.io.*;
import java.util.Arrays;

class IOEx1 {
    public static void main(String[] args) {
        byte[] inSrc = {0,1,2,3,4,5,6,7,8,9};
        byte[] outSrc = null;

        ByteArrayInputStream input = null;
        ByteArrayOutputStream output = null;

        input = new ByteArrayInputStream(inSrc);
        output = new ByteArrayOutputStream();

        int data = 0;

        while((data = input.read()) != -1) {
            output.write(data); // void write(int b)
        }

        outSrc = output.toByteArray(); // 스트림의 내용을 byte배열로 반환한다.

        System.out.println("Input Source :" + Arrays.toString(inSrc));
        System.out.println("Output Source :" + Arrays.toString(outSrc));
    }
}
```

(data = input.read()) != -1
① data = input.read() // read()를 호출한 반환값을 변수 data에 저장한다.
② data != -1 // data에 저장된 값이 -1이 아님지 비교한다.

abstract int read()

1 byte를 읽어 온다(0~255사이의 값).
더 이상 읽어 올 데이터가 없으면 -1
을 반환한다.

【실행결과】

Input Source :[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Output Source :[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

2.2 ByteArrayInputStream과 ByteArrayOutputStream

```
import java.io.*;
import java.util.Arrays;

class IOEx3 {
    public static void main(String[] args) {
        byte[] inSrc = {0,1,2,3,4,5,6,7,8,9};
        byte[] outSrc = null;

        byte[] temp = new byte[4]; // 이전 예제와 배열의 크기가 다르다.

        ByteArrayInputStream input = null;
        ByteArrayOutputStream output = null;

        input = new ByteArrayInputStream(inSrc);
        output = new ByteArrayOutputStream();

        try {
            while(input.available() > 0) {
                input.read(temp);
                output.write(temp);
            }
        } catch(IOException e) {}

        outSrc = output.toByteArray();

        System.out.println("Input Source :" + Arrays.toString(inSrc));
        System.out.println("temp      :" + Arrays.toString(temp));
        System.out.println("Output Source :" + Arrays.toString(outSrc));
    }
}
```

[실행결과]

```
Input Source :[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
temp       :[8, 9, 6, 7]
Output Source :[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 6, 7]
```

[실행결과]

```
Input Source :[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
temp       :[8, 9, 6, 7]
Output Source :[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
int len = input.read(temp);
output.write(temp, 0, len);
```

2.3 FileInputStream과 FileOutputStream

- 파일(file)에 데이터를 입출력하는 바이트기반 스트림

생성자	설명
FileInputStream(String name)	지정된 파일이름(name)을 가진 실제 파일과 연결된 FileInput Stream을 생성한다.
FileInputStream(File file)	파일의 이름이 String이 아닌 File인스턴스로 지정해주어야 하는 점을 제외하고 FileInputStream(String name)와 같다.
FileOutputStream(String name)	지정된 파일이름(name)을 가진 실제 파일과의 연결된 File OutputStream을 생성한다.
FileOutputStream(String name, boolean append)	지정된 파일이름(name)을 가진 실제 파일과 연결된 File OutputStream을 생성한다. 두번째 인자인 append를 true로 하면, 출력 시 기존의 파일내용의 마지막에 덧붙인다. false면, 기존의 파일내용을 덮어쓰게 된다.
FileOutputStream(File file)	파일의 이름을 String이 아닌 File인스턴스로 지정해주어야 하는 점을 제외하고 FileOutputStream(String name)과 같다.

```
import java.io.*;

class FileCopy {
    public static void main(String args[]) {
        try {
            FileInputStream fis = new FileInputStream(args[0]);
            FileOutputStream fos = new FileOutputStream(args[1]);

            int data =0;
            while((data=fis.read())!=-1) {
                fos.write(data);      // void write(int b)
            }

            fis.close();
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

【실행결과】

```
C:\jdk1.5\work\ch14>java FileCopy FileCopy.java FileCopy.bak
```

```
C:\jdk1.5\work\ch14>
```

3. 바이트기반 보조스트림

3.1 FilterInputStream과 FilterOutputStream

- 모든 바이트기반 보조스트림의 최고조상
- 보조스트림은 자체적으로 입출력을 수행할 수 없다.

```
protected FilterInputStream(InputStream in)
public FilterOutputStream(OutputStream out)
```

- 상속을 통해 FilterInputStream/FilterOutputStream의 read()와 write()를 원하는 기능대로 오버라이딩해야 한다.

```
public class FilterInputStream extends InputStream {
    protected volatile InputStream in;
    protected FilterInputStream(InputStream in) {
        this.in = in;
    }

    public int read() throws IOException {
        return in.read();
    }
    ...
}
```

FilterInputStream의 자손 - BufferedInputStream, DataInputStream, PushbackInputStream 등
FilterOutputStream의 자손 - BufferedOutputStream, DataOutputStream, PrintStream 등

3.2 BufferedInputStream과 BufferedOutputStream

- 입출력 효율을 높이기 위해 버퍼(byte[])를 사용하는 보조스트림

메서드 / 생성자	설명
BufferedInputStream(InputStream in, int size)	주어진 InputStream인스턴스를 입력소스(input source)로하여 지정된 크기(byte 단위)의 버퍼를 갖는 BufferedInputStream인스턴스를 생성한다.
BufferedInputStream(InputStream in)	주어진 InputStream인스턴스를 입력소스(input source)로하여 버퍼의 크기를 지정해주지 않으므로 기본적으로 8192 byte 크기의 버퍼를 갖게 된다.

메서드 / 생성자	설명
BufferedOutputStream(OutputStream out, int size)	주어진 OutputStream인스턴스를 출력소스(output source)로하여 지정된 크기(단위byte)의 버퍼를 갖는 BufferedOutputStream인스턴스를 생성한다.
BufferedOutputStream(OutputStream out)	주어진 OutputStream인스턴스를 출력소스(output source)로하여 버퍼의 크기를 지정해주지 않으므로 기본적으로 8192 byte 크기의 버퍼를 갖게 된다.
flush()	버퍼의 모든 내용을 출력소스에 출력한 다음, 버퍼를 비운다.
close()	flush()를 호출해서 버퍼의 모든 내용을 출력소스에 출력하고, BufferedOutputStream인스턴스가 사용하던 모든 자원을 반환한다.

- 보조스트림을 닫으면 기반스트림도 닫힌다.

```
public class FilterOutputStream extends OutputStream {  
    protected OutputStream out;  
    public FilterOutputStream(OutputStream out) {  
        this.out = out;  
    }  
    ...  
    public void close() throws IOException {  
        try { flush(); } catch (IOException ignored) {}  
        out.close(); // 기반 스트림의 close()를 호출한다.  
    }  
}
```

3.2 BufferedInputStream과 BufferedOutputStream

```
import java.io.*;

class BufferedOutputStreamEx1 {
    public static void main(String args[]) {
        try {
            FileOutputStream fos = new FileOutputStream("123.txt");
            // BufferedOutputStream의 버퍼 크기를 5로 한다.
            BufferedOutputStream bos = new BufferedOutputStream(fos, 5);
            // 파일 123.txt에 1부터 9까지 출력한다.
            for(int i='1'; i <= '9'; i++) {
                bos.write(i);
            }

            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

자바 프로그램

BufferedOutputStreamEx1

bos.write(i);

BufferedOutputStream



FileOutputStream

123.txt

[실행결과]

C:\jdk1.5\work\ch14>java BufferedOutputStreamEx1

C:\jdk1.5\work\ch14>type 123.txt

12345

3.3 DataInputStream과 DataOutputStream

- 기본형 단위로 읽고 쓰는 보조스트림
- 각 자료형의 크기가 다르므로 출력할 때와 입력할 때 순서에 주의

메서드 / 생성자	설명
DataInputStream(InputStream in)	주어진 InputStream인스턴스를 기반스트림으로 하는 DataInputStream인스턴스를 생성한다.
boolean readBoolean() byte readByte() char readChar() short readShort() int readInt() long readLong() float readFloat() double readDouble()	각 자료형에 알맞은 값을 읽어온다. 더 이상 읽을 값이 없으면 EOFException을 발생시킨다.
String readUTF()	UTF형식으로 쓰여진 문자를 읽는다. 더 이상 읽을 값이 없으면 EOFException을 발생시킨다.
int skipBytes(int n)	현재 읽고 있는 위치에서 지정된 숫자(n) 만큼을 건너뛴다.

메서드 / 생성자	설명
DataOutputStream(OutputStream out)	주어진 OutputStream인스턴스를 기반스트림으로 하는 DataOutputStream인스턴스를 생성한다.
void writeBoolean(boolean b) void writeByte(int b) void writeChar(int c) void writeShort(int s) void writeInt(int l) void writeLong(long l) void writeFloat(float f) void writeDouble(double d)	각 자료형에 알맞은 값을 출력한다.
void writeUTF(String s)	UTF형식으로 문자를 출력한다.
void writeChars(String s)	주어진 문자열을 출력한다. writeChar(char c)메서드를 여러 번 호출한 결과와 같다.
int size()	지금까지 DataOutputStream에 쓰여진 byte의 수를 알려준다.

3.4 SequenceInputStream

- 여러 입력스트림을 연결해서 하나의 스트림처럼 다룰 수 있게 해준다.

메서드 / 생성자	설명
SequenceInputStream(Enumeration e)	Enumeration에 저장된 순서대로 입력스트림을 하나의 스트림으로 연결한다.
SequenceInputStream(InputStream s1, InputStream s2)	두 개의 입력스트림을 하나로 연결한다.

【사용예1】

```
Vector files = new Vector();
files.add(new FileInputStream("file.001"));
files.add(new FileInputStream("file.002"));
SequenceInputStream in = new SequenceInputStream(files.elements());
```

【사용예2】

```
FileInputStream file1 = new FileInputStream("file.001");
FileInputStream file2 = new FileInputStream("file.002");
SequenceInputStream in = new SequenceInputStream(file1, file2);
```

3.5 PrintStream (1/2)

- 데이터를 다양한 형식의 문자로 출력하는 기능을 제공하는 보조스트림
- System.out과 System.err이 PrintStream이다.
- PrintStream보다 PrintWriter를 사용할 것을 권장한다.

생성자 / 메서드	설명
PrintStream(File file) PrintStream(File file, String csn) PrintStream(OutputStream out) PrintStream(OutputStream out,boolean autoFlush) PrintStream(OutputStream out,boolean autoFlush, String encoding) PrintStream(String fileName) PrintStream(String fileName, String csn)	지정된 출력스트림을 기반으로 하는 PrintStream 인스턴스를 생성한다. autoFlush의 값을 true로 하면 println메서드가 호출되거나 개행문자가 출력될 때 자동으로 flush된다. 기본값은 false이다.
boolean checkError()	스트림을 flush하고 에러가 발생했는지를 알려 준다.
void print(boolean b) void print(char c) void print(char[] c) void print(double d) void print(float f) void print(int i) void print(long l) void print(Object o) void print(String s)	인자로 주어진 값을 출력소스에 문자로 출력한다. println메서드는 출력 후 줄바꿈을 하고, print메서드는 줄을 바꾸지 않는다.
void println()	줄바꿈 문자(line separator)를 출력함으로써 줄을 바꾼다.
PrintStream printf(String format, Object... args)	정형화된(formatted) 출력을 가능하게 한다.
protected void setError()	작업 중에 오류가 발생했음을 알린다.(setError()를 호출한 후에, checkError()를 호출하면 true를 반환한다.)

3.5 PrintStream (2/2)

format	설명	결과(int i=65)
%d	10진수(decimal integer)	65
%o	8진수(octal integer)	101
%x	16진수(hexadecimal integer)	41
%c	문자	A
%s	문자열	65
%5d	5자리 숫자. 빈자리는 공백으로 채운다.	65
%-5d	5자리 숫자. 빈자리는 공백으로 채운다.(왼쪽 정렬)	65
%05d	5자리 숫자. 빈자리는 0으로 채운다.	00065

format	설명	결과
%e	지수형태표현(exponent)	1.234568e+03
%f	10진수(decimal float)	1234.56789
%3.1f	출력될 자리수를 최소 3자리(소수점포함), 소수점 이하 1자리(2번째 자리에서 반올림)	1234.6
%8.1f	소수점이상 최소 6자리, 소수점 이하 1자리. 출력될 자리수를 최소 8자리(소수점포함)를 확보한다. 빈자리는 공백으로 채워진다.(오른쪽 정렬)	1234.6
%08.1f	소수점이상 최소 6자리, 소수점 이하 1자리. 출력될 자리수를 최소 8자리(소수점포함)를 확보한다. 빈자리는 0으로 채워진다.	001234.6
%-8.1f	소수점이상 최소 6자리, 소수점 이하 1자리. 출력될 자리수를 최소 8자리(소수점포함)를 확보한다. 빈자리는 공백으로 채워진다.(왼쪽 정렬) 1234.6	1234.6

format	설명	결과
%s	문자열(string)	ABC
%5s	5자리 문자열. 빈자리는 공백으로 채운다.	ABC
%-5s	5자리 문자열. 빈자리는 공백으로 채운다.(왼쪽 정렬)	ABC

format	설명
\t	탭(tab)
\n	줄바꿈 문자(new line)
% %	%

format	설명	결과
%tR %tH:%tM	시분(24시간)	21:05 21:05
%tT %tH:%tM:%tS	시분초(24시간)	21:05:33 21:05:33
%tD %tm/%td/%ty	연월일	02/16/07 02/16/07
%tF %tY-%tm-%td	연월일	2007-02-16 2007-02-16

4. 문자기반 스트림

4.1 Reader와 Writer

▶ Reader(문자기반 입력스트림의 최고 조상)의 메서드

메서드	설명
abstract void close()	입력스트림을 닫음으로써 사용하고 있던 자원을 반환한다.
void mark(int readlimit)	현재위치를 표시해놓는다. 후에 reset()에 의해서 표시해 놓은 위치로 다시 돌아갈 수 있다.
boolean markSupported()	mark()와 reset()을 지원하는지를 알려 준다.
int read()	입력소스로부터 하나의 문자를 읽어 온다. char의 범위인 0~65535범위의 정수를 반환하며, 입력스트림의 마지막 데이터에 도달하면, -1을 반환한다.
int read(char[] c);	입력소스로부터 매개변수로 주어진 배열 c의 크기만큼 읽어서 배열 c에 저장한다. 읽어 온 데이터의 개수 또는 -1을 반환한다.
abstract int read(char[] c, int off, int len)	입력소스로부터 최대 len개의 문자를 읽어서, 배열 c의 지정된 위치(off)부터 읽은 만큼 저장한다. 읽어 온 데이터의 개수 또는 -1을 반환한다.
boolean ready()	입력소스로부터 데이터를 읽을 준비가 되어있는지 알려 준다.
void reset()	입력소스에서의 위치를 마지막으로 mark()가 호출되었던 위치로 되돌린다.
long skip(long n)	현재 위치에서 주어진 문자 수(n)만큼을 건너뛴다.

▶ Writer(문자기반 출력스트림의 최고 조상)의 메서드

메서드	설명
abstract void close()	출력스트림을 닫음으로써 사용하고 있던 자원을 반환한다.
abstract void flush()	스트림의 버퍼에 있는 모든 내용을 출력소스에 쓴다.(버퍼가 있는 스트림에만 해당됨)
void write(int b)	주어진 값을 출력소스에 쓴다.
void write(char[] c)	주어진 배열 c에 저장된 모든 내용을 출력소스에 쓴다.
abstract void write(char[] c, int off, int len)	주어진 배열 c에 저장된 내용 중에서 off번째부터 len길이만큼만 출력소스에 쓴다.
void write(String str)	주어진 문자열(str)을 출력소스에 쓴다.
void write(String str, int off, int len)	주어진 문자열(str)의 일부를 출력소스에 쓴다.(off번째 문자부터 len개 만큼의 문자열)

4.2 FileReader와 FileWriter

- 문자기반의 파일 입출력. 텍스트 파일의 입출력에 사용한다.

```
import java.io.*;

class FileReaderEx1 {
    public static void main(String args[]) {
        try {
            String fileName = "test.txt";
            FileInputStream fis = new FileInputStream(fileName);
            FileReader fr = new FileReader(fileName);

            int data = 0;
            // FileInputStream을 이용해서 파일내용을 읽어 화면에 출력한다.
            while((data=fis.read())!=-1) {
                System.out.print((char)data);
            }
            System.out.println();
            fis.close();

            // FileReader를 이용해서 파일내용을 읽어 화면에 출력한다.
            while((data=fr.read())!=-1) {
                System.out.print((char)data);
            }
            System.out.println();
            fr.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    } // main
}
```

[실행결과]

```
C:\jdk1.5\work\ch14>type test.txt
Hello, 안녕하세요?
```

```
C:\jdk1.5\work\ch14>java FileReaderEx1
Hello, ½?°????¾???
Hello, 안녕하세요?
```

4.3 PipedReader와 PipedWriter

- 프로세스(쓰레드)간의 통신(데이터를 주고 받음)에 사용한다.

```
class InputThread extends Thread {  
    PipedReader input = new PipedReader();  
    StringWriter sw = new StringWriter();  
  
    InputThread(String name) { super(name); }  
  
    public void run() {  
        try {  
            int data = 0;  
  
            while((data=input.read()) != -1) {  
                sw.write(data);  
            }  
            System.out.println(getName()  
                + " received : " + sw.toString());  
        } catch(IOException e) {}  
    } // run  
  
    public PipedReader getInput() { return input; }  
    public void connect(PipedWriter output) {  
        try {  
            input.connect(output);  
        } catch(IOException e) {}  
    } // connect  
}
```

【실행결과】

```
OutputThread sent : Hello  
InputThread received : Hello
```

```
class OutputThread extends Thread {  
    PipedWriter output = new PipedWriter();  
  
    OutputThread(String name) { super(name); }  
  
    public void run() {  
        try {  
            String msg = "Hello";  
            System.out.println(getName()  
                + " sent : " + msg);  
            output.write(msg);  
            output.close();  
        } catch(IOException e) {}  
    } // run  
  
    public PipedWriter getOutput() { return output; }  
    public void connect(PipedReader input) {  
        try {  
            output.connect(input);  
        } catch(IOException e) {}  
    } // connect  
}
```

```
public static void main(String args[]) {  
    InputThread inThread = new InputThread("InputThread");  
    OutputThread outThread = new OutputThread("OutputThread");  
    //PipedReader와 PipedWriter를 연결한다.  
    inThread.connect(outThread.getOutput());  
    inThread.start(); outThread.start();  
} // main
```

4.4 StringReader와 StringWriter

- CharArrayReader, CharArrayWriter처럼 메모리의 입출력에 사용한다.
- StringWriter에 출력되는 데이터는 내부의 StringBuffer에 저장된다.

```
StringBuffer getBuffer() : StringWriter에 출력한 데이터가 저장된 StringBuffer를 반환한다.  
String toString() : StringWriter에 출력된 (StringBuffer에 저장된) 문자열을 반환한다.
```

```
import java.io.*;  
  
class StringReaderWriterEx {  
    public static void main(String[] args) {  
        String inputData = "ABCD";  
        StringReader input = new StringReader(inputData);  
        StringWriter output = new StringWriter();  
  
        int data = 0;  
  
        try {  
            while((data = input.read())!=-1) {  
                output.write(data); // void write(int b)  
            }  
        } catch(IOException e) {}  
  
        System.out.println("Input Data :" + inputData);  
        System.out.println("Output Data :" + output.toString());  
        // System.out.println("Output Data :" + output.getBuffer().toString());  
    }  
}
```

[실행결과]

```
Input Data :ABCD  
Output Data :ABCD
```

5. 문자기반 보조스트림

5.1 BufferedReader와 BufferedWriter

- 입출력 효율을 높이기 위해 버퍼(char[])를 사용하는 보조스트림
- 라인(line)단위의 입출력이 편리하다.

```
String readLine() - 한 라인을 읽어온다. (BufferedReader의 메서드)  
void newLine() - '라인 구분자(개행문자)'를 출력한다. (BufferedWriter의 메서드)
```

```
import java.io.*;  
  
class BufferedReaderEx1 {  
    public static void main(String[] args) {  
        try {  
            FileReader fr = new FileReader("BufferedReaderEx1.java");  
            BufferedReader br = new BufferedReader(fr);  
  
            String line = "";  
            for(int i=1;(line = br.readLine())!=null;i++) {  
                // ";"를 포함한 라인을 출력한다.  
                if(line.indexOf(";")!=-1)  
                    System.out.println(i+":"+line);  
            }  
            br.close();  
        } catch(IOException e) {}  
    } // main  
}
```

[실행결과]

```
1:import java.io.*;  
6:      FileReader fr = new FileReader("BufferedReaderEx1.java");  
7:      BufferedReader br = new BufferedReader(fr);  
9:      String line = "";  
10:     for(int i=1;(line = br.readLine())!=null;i++) {  
11:         // ";"를 포함한 라인을 출력한다.  
12:         if(line.indexOf(";")!=-1)  
13:             System.out.println(i+":"+line);
```

5.2 InputStreamReader와 OutputStreamWriter

- 바이트기반스트림을 문자기반스트림처럼 쓸 수 있게 해준다.
- 인코딩(encoding)을 변환하여 입출력할 수 있게 해준다.

생성자 / 메서드	설명
InputStreamReader(InputStream in)	OS에서 사용하는 기본 인코딩의 문자로 변환하는 InputStreamReader를 생성한다.
InputStreamReader(InputStream in, String encoding)	지정된 인코딩을 사용하는 InputStreamReader를 생성한다.
String getEncoding()	InputStreamReader의 인코딩을 알려 준다.

생성자 / 메서드	설명
OutputStreamWriter(OutputStream in)	OS에서 사용하는 기본 인코딩의 문자로 변환하는 OutputStreamWriter를 생성한다.
OutputStreamWriter(OutputStream in, String encoding)	지정된 인코딩을 사용하는 OutputStreamWriter를 생성한다.
String getEncoding()	OutputStreamWriter의 인코딩을 알려 준다.

- 콘솔(console, 화면)로부터 라인단위로 입력받기

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String line = br.readLine();
```

- 인코딩 변환하기

```
Properties prop = System.getProperties();
System.out.println(prop.get("sun.jnu.encoding"));
```

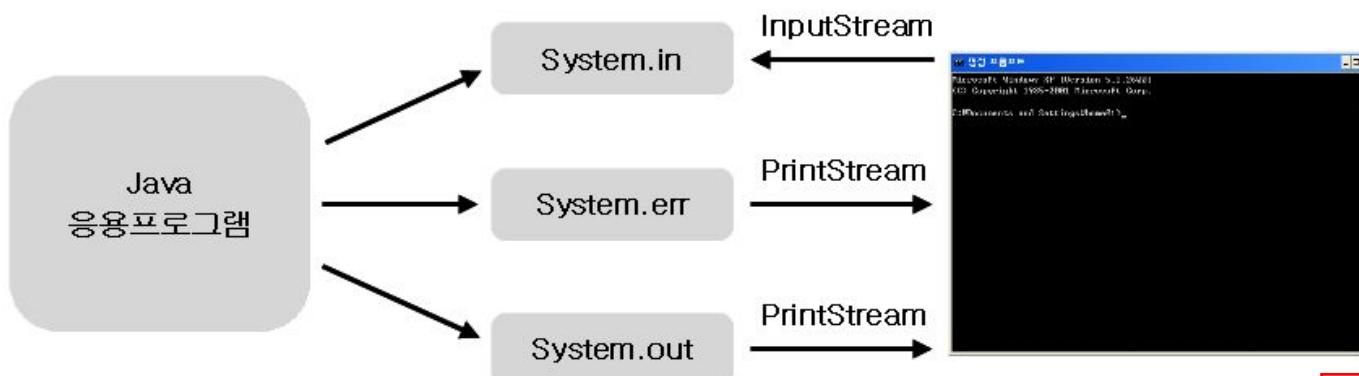
```
FileInputStream fis = new FileInputStream("korean.txt");
InputStreamReader isr = new InputStreamReader(fis, "KSC5601");
```

6. 표준입출력과 File

6.1 표준입출력 – System.in, System.out, System.err

- 콘솔(console, 화면)을 통한 데이터의 입출력을 ‘표준 입출력’이라 한다.
- JVM이 시작되면서 자동적으로 생성되는 스트림이다.

System.in - 콘솔로부터 데이터를 입력받는데 사용
System.out - 콘솔로 데이터를 출력하는데 사용
System.err - 콘솔로 데이터를 출력하는데 사용



```
public final class System {  
    public final static InputStream in = nullInputStream();  
    public final static PrintStream out = nullPrintStream();  
    public final static PrintStream err = nullPrintStream();  
    ...  
}
```

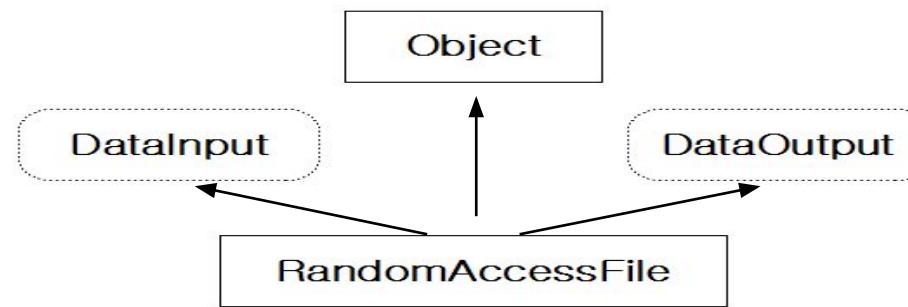
```
static void setOut(PrintStream out)  
static void setErr(PrintStream err)  
static void setIn(InputStream in)
```

6.2 RandomAccessFile

- 하나의 스트림으로 파일에 입력과 출력을 모두 수행할 수 있는 스트림
- 다른 스트림들과 달리 Object의 자손이다.

* DataInput 인터페이스의 메서드

```
boolean readBoolean()  
byte   readByte()  
int    readInt()  
void   readFully(byte[] b)  
String readLine()  
...  
...
```



* DataOutput 인터페이스의 메서드

```
void write(byte[] b)  
void write(int b)  
void writeBoolean(boolean b)  
void writeInt(int v)  
void writeBytes(String s)  
...  
...
```

생성자 / 메서드	설명
RandomAccessFile(File file, String mode) RandomAccessFile(String fileName, String mode)	주어진 file에 읽기 또는 읽기와 쓰기를 하기 위한 RandomAccessFile인스턴스를 생성한다. mode에는 "r"과 "rw" 두 가지 값이 지정 가능하다. "r" - 파일로부터 읽기(r)만을 수행할 때 "rw" - 파일에 읽기(r)와 쓰기(w)
long getFilePointer()	파일 포인터의 위치를 알려 준다.
long length()	파일의 크기를 얻을 수 있다.(단위 byte)
void seek(long pos)	파일 포인터의 위치를 변경한다. 위치는 파일의 첫 부분부터 pos크기만큼 떨어진 곳이다.(단위 byte)
void setLength(long newLength)	파일의 크기를 지정된 길이로 변경한다.(byte 단위)
int skipBytes(int n)	지정된 수만큼의 byte를 건너뛴다.

6.3 File (1/2) – 생성자와 경로관련 메서드

- 파일과 디렉토리를 다루는데 사용되는 클래스

생성자 / 메서드	설명
File(String fileName)	주어진 문자열(fileName)을 이름으로 갖는 파일을 위한 File인스턴스를 생성한다. 파일 뿐만 아니라 디렉토리도 같은 방법으로 다룬다. 여기서 fileName은 주로 경로(path)를 포함해서 지정해주지만, 파일 이름만 사용해도 되는데 이 경우 프로그램이 실행되는 위치가 경로(path)로 간주된다.
File(String pathName, String fileName) File(File pathName, String fileName)	파일의 경로와 이름을 따로 분리해서 지정할 수 있도록 한 생성자. 이 중 두 번째 것은 경로를 문자열이 아닌 File인스턴스인 경우를 위해서 제공된 것이다.
String getName()	파일이름을 String으로 반환한다.
String getPath()	파일의 경로(path)를 String으로 반환한다.
String getAbsolutePath() File getAbsoluteFile()	파일의 절대경로를 String으로 반환한다. 파일의 절대경로를 File로 반환한다.
String getParent() File getParentFile()	파일의 조상 디렉토리를 String으로 반환한다. 파일의 조상 디렉토리를 File로 반환한다.
String getCanonicalPath() File getCanonicalFile()	파일의 정규경로를 String으로 반환한다. 파일의 정규경로를 File로 반환한다.

멤버변수	설명
static String pathSeparator	OS에서 사용하는 경로(path) 구분자. 윈도우 ";", 유닉스 ":"
static char pathSeparatorChar	OS에서 사용하는 경로(path) 구분자. 윈도우에서는 ';', 유닉스 '::'
static String separator	OS에서 사용하는 이름 구분자. 윈도우 "₩", 유닉스 "/"
static char separatorChar	OS에서 사용하는 이름 구분자. 윈도우 '₩', 유닉스 '/'

6.3 File (1/2) – 생성자와 경로관련 메서드(예제)

```
File f = new File("c:\\jdk1.5\\work\\ch14\\FileEx1.java");
String fileName = f.getName();
int pos = fileName.lastIndexOf(".");
```

경로를 제외한 파일이름 - FileEx1.java
확장자를 제외한 파일이름 - FileEx1
확장자 - java

```
System.out.println("경로를 제외한 파일이름 - " + f.getName());
System.out.println("확장자를 제외한 파일이름 - " + fileName.substring(0, pos));
System.out.println("확장자 - " + fileName.substring(pos+1));
```

```
System.out.println("경로를 포함한 파일이름 - " + f.getPath());
System.out.println("파일의 절대경로 - " + f.getAbsolutePath());
System.out.println("파일이 속해 있는 디렉토리 - " + f.getParent());
```

```
File.separator - ;
File.separatorChar - ;
File.separator - \
File.separatorChar - \
    tln("File.separator - " + File.separator);
System.out.println("File.separatorChar - " + File.separatorChar);
System.out.println("File.separator - " + File.separator);
System.out.println("File.separatorChar - " + File.separatorChar);
```

경로를 포함한 파일이름 - c:\\jdk1.5\\work\\ch14\\FileEx1.java
파일의 절대경로 - c:\\jdk1.5\\work\\ch14\\FileEx1.java
파일의 정규경로 - C:\\jdk1.5\\work\\ch14\\FileEx1.java
파일이 속해 있는 디렉토리 - c:\\jdk1.5\\work\\ch14

```
System.out.println("user.dir="+System.getProperty("user.dir"));
System.out.println("sun.boot.class.path="+ System.getProperty("sun.boot.class.path"));
```

user.dir=C:\\jdk1.5\\work\\ch14
sun.boot.class.path=C:\\jdk1.5\\jre\\lib\\rt.jar;C:\\jdk1.5\\jre\\lib\\i18n.jar;C:\\j
dk1.5\\jre\\lib\\sunrsasign.jar;C:\\jdk1.5\\jre\\lib\\jsse.jar;C:\\jdk1.5\\jre\\lib\\jc
e.jar;C:\\jdk1.5\\jre\\lib\\charsets.jar;C:\\jdk1.5\\jre\\classes

6.3 File (2/2) – 파일의 속성, 생성, 삭제, 목록

메서드	설명
boolean canRead()	읽을 수 있는 파일인지 검사한다.
boolean canWrite()	쓸 수 있는 파일인지 검사한다.
boolean exists()	파일이 존재하는지 검사한다.
boolean isAbsolute()	파일 또는 디렉토리가 절대경로명으로 지정되었는지 확인한다.
boolean isDirectory()	디렉토리인지 확인한다.
boolean isFile()	파일인지 확인한다.
boolean isHidden()	파일의 속성이 '숨김(Hidden)'인지 확인한다. 또한 파일이 존재하지 않으면 false를 반환한다.
int compareTo(File pathname)	주어진 파일 또는 디렉토리를 비교한다. 같으면 0을 반환하며, 다르면 1 또는 -1을 반환한다. (Unix시스템에서는 대소문자를 구별하며, Windows에서는 구별하지 않는다.)
boolean createNewFile()	아무런 내용이 없는 새로운 파일을 생성한다.(파일이 이미 존재하면 생성되지 않는다.) File f = new File("c:\jdk1.5\work\test3.java"); f.createNewFile();
static File createTempFile(String prefix, String suffix)	임시 파일을 시스템의 임시 디렉토리에 생성한다. System.out.println(File.createTempFile("work", ".tmp")); 결과 : c:\windows\TEMP\work14247.tmp
static File createTempFile(String prefix, String suffix, File directory)	임시 파일을 시스템의 지정된 디렉토리에 생성한다.
boolean delete()	파일을 삭제한다.
void deleteOnExit()	응용 프로그램 종료시 파일을 삭제한다. 주로 임시 파일을 삭제하는데 사용된다.
boolean equals(Object obj)	주어진 객체(주로 File인스턴스)가 같은 파일인지 비교한다. (Unix시스템에서는 대소문자를 구별하며, Windows에서는 구별하지 않는다.)
long length()	파일의 크기를 반환한다.
String [] list()	디렉토리의 파일 목록(디렉토리 포함)을 String 배열로 반환한다.
String [] list(FilenameFilter filter)	FilenameFilter 인스턴스에 구현된 조건에 맞는 파일을 String 배열로 반환한다.
File [] listFiles()	디렉토리의 파일 목록(디렉토리 포함)을 File 배열로 반환한다.
static File [] listRoots()	컴퓨터의 파일 시스템의 root의 목록(floppy, CD-ROM, HDD drive)을 반환한다. (예: A:\, C:\, D:\)

6.3 File (2/2) – 파일의 속성, 생성, 삭제, 목록(예제1)

```
import java.io.*;  
  
class FileEx2 {  
    public static void main(String[] args)  
    {  
        if(args.length != 1) {  
            System.out.println("USAGE : java FileEx2 DIRECTORY");  
            System.exit(0);  
        }  
  
        File f = new File(args[0]);  
  
        if(!f.exists() || !f.isDirectory()) {  
            System.out.println("유효하지 않은 디렉토리입니다.");  
            System.exit(0);  
        }  
  
        File[] files = f.listFiles();  
  
        for(int i=0; i < files.length; i++) {  
            String fileName = files[i].getName();  
            System.out.println(  
                files[i].isDirectory() ? "["+fileName+"]" : fileName);  
        }  
    } // main  
}
```

【실행결과】

```
C:\jdk1.5\work\ch14>java FileEx2  
USAGE : java FileEx2 DIRECTORY  
  
C:\jdk1.5\work\ch14>java FileEx2 work  
유효하지 않은 디렉토리입니다.  
  
C:\jdk1.5\work\ch14>java FileEx2 c:\jdk1.5  
[bin]  
COPYRIGHT  
[demo]  
[docs]  
[include]  
jdk-1_5_0-doc.zip  
[jre]  
[lib]  
LICENSE  
... 중간생략 ...  
  
C:\jdk1.5\work\ch14>
```

6.3 File (2/2) – 파일의 속성, 생성, 삭제, 목록(예제2)

```
public static void print fileList(File dir) {  
    System.out.println(dir.getAbsolutePath() + " 디렉토리");  
    File[] files = dir.listFiles();  
  
    ArrayList subDir = new ArrayList();  
  
    for(int i=0; i < files.length; i++) {  
        String filename = files[i].getName();  
  
        if(files[i].isDirectory()) {  
            filename = "[" + filename + "]";  
            subDir.add(i++);  
        }  
        System.out.println(filename);  
    }  
  
    int dirNum = subDir.size();  
    int fileNum = files.length - dirNum;  
  
    totalFiles += fileNum;  
    totalDirs += dirNum;  
  
    System.out.println(fileNum + "개의 파일, " + dirNum + "개의 디렉토리");  
    System.out.println();  
  
    for(int i=0; i < subDir.size(); i++) {  
        int index = Integer.parseInt((String)subDir.get(i));  
        print fileList(files[index]);  
    }  
} // print fileList
```

```
C:\jdk1.5\work\ch14>java FileEx3 c:\jdk1.5\work\ch14  
c:\jdk1.5\work\ch14 디렉토리  
FileEx1.java  
FileEx2.class  
FileEx2.java  
...  
VectorEx2.java  
20개의 파일, 2개의 디렉토리  
  
c:\jdk1.5\work\ch14\temp 디렉토리  
FileEx9.class  
FileEx9.java  
FileEx9.java.bak  
result.txt  
[temp]  
[temp]  
[temp]  
4개의 파일, 2개의 디렉토리
```

6.3 File (2/2) – 파일의 속성, 생성, 삭제, 목록(예제3)

```
public static void main(String[] args) {
    String currDir = System.getProperty("user.dir");
    File dir = new File(currDir);

    File[] files = dir.listFiles();

    for(int i=0; i < files.length; i++) {
        File f = files[i];
        String name = f.getName();
        SimpleDateFormat df =new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String attribute = "";
        String size = "";

        if(files[i].isDirectory()) {
            attribute = "DIR";
        } else {
            size = f.length() + "";
            attribute = f.canRead() ? "R" : " ";
            attribute += f.canWrite() ? "W" : " ";
            attribute += f.isHidden() ? "H" : " ";
        }

        System.out.printf("%s %3s %6s %s\n",df.format(new Date(f.lastModified()))
                        , attribute, size, name );
    }
}
```

[실행결과]

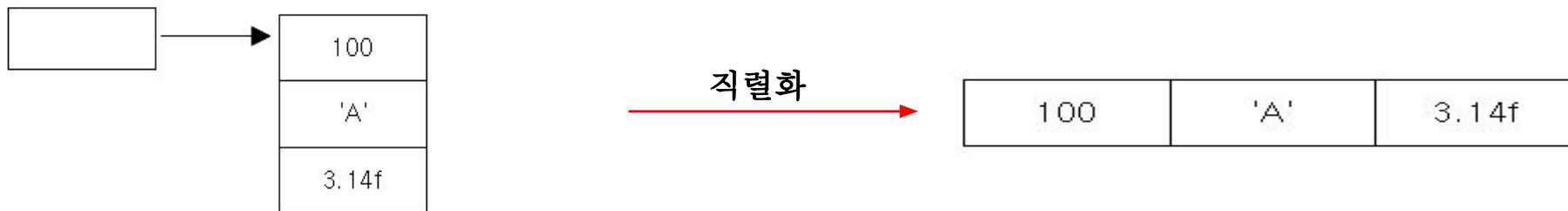
```
C:\jdk1.5\work\ch14>java FileEx4
2006-11-20 16:39오후 RW    1484 FileEx4.class
2006-11-20 16:39오후 RW    2171 FileEx4.java
2006-11-20 16:38오후 RW    2170 FileEx4.java.bak
...
2006-07-27 17:10오후 DIR      Temp

C:\jdk1.5\work\ch14>
```

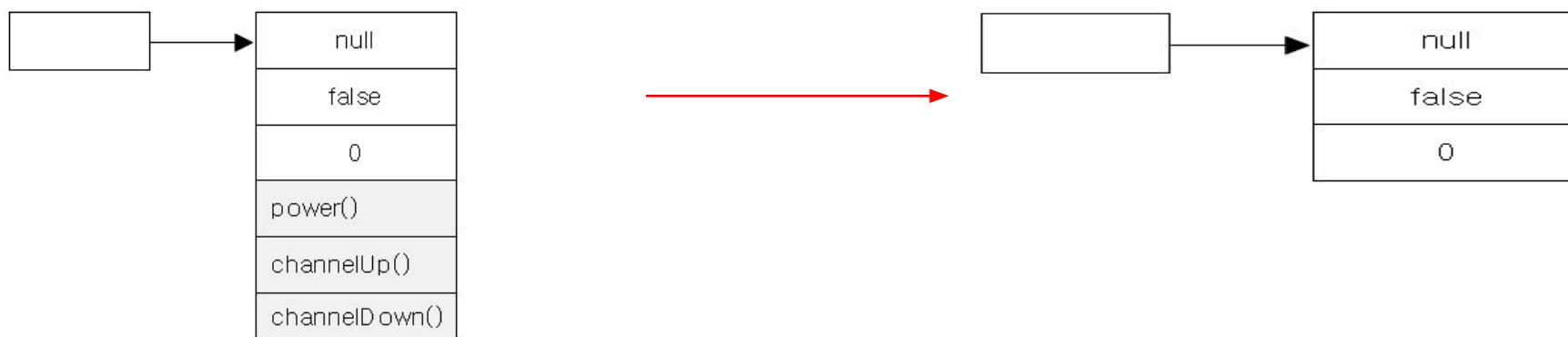
7. 직렬화(Serialization)

7.1 직렬화(serialization)란?

- 객체를 ‘연속적인 데이터’로 변환하는 것. 반대과정은 ‘역직렬화’라고 한다.
- 객체의 인스턴스변수들의 값을 일렬로 나열하는 것



- 객체를 저장하기 위해서는 객체를 직렬화해야 한다.
- 객체를 저장한다는 것은 객체의 모든 인스턴스변수의 값을 저장하는 것



7.2 ObjectInputStream, ObjectOutputStream

- 객체를 직렬화하여 입출력할 수 있게 해주는 보조스트림

```
ObjectInputStream(InputStream in)
ObjectOutputStream(OutputStream out)
```

- 객체를 파일에 저장하는 방법

```
FileOutputStream fos = new FileOutputStream("objectfile.ser");
ObjectOutputStream out = new ObjectOutputStream(fos);

out.writeObject(new UserInfo());
```

- 파일에 저장된 객체를 다시 읽어오는 방법

```
FileInputStream fis = new FileInputStream("objectfile.ser");
ObjectInputStream in= new ObjectInputStream(fis);

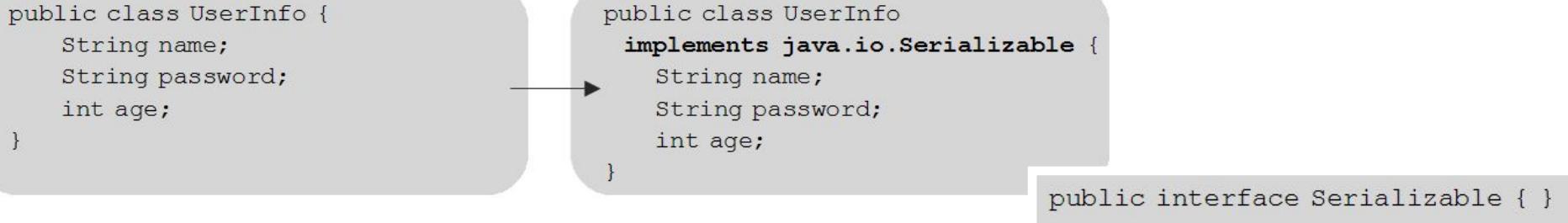
UserInfo info = (UserInfo)in.readObject();
```

ObjectOutputStream
void defaultWriteObject()
void write(byte[] buf)
void write(byte[] buf, int off, int len)
void write(int val)
void writeBoolean(boolean val)
void writeByte(int val)
void writeBytes(String str)
void writeChar(int val)
void writeChars(String str)
void writeDouble(double val)
void writeFloat(float val)
void writeInt(int val)
void writeLong(long val)
void writeObject(Object obj)
void writeShort(int val)
void writeUTF(String str)

ObjectInputStream
void defaultReadObject()
int read()
int read(byte[] buf, int off, int len)
boolean readBoolean()
byte readByte()
char readChar()
double readDouble()
float readFloat()
int readInt()
long readLong()
short readShort()
Object readObject()
String readUTF

7.3 직렬화 가능한 클래스 만들기(1/2)

- `java.io.Serializable`을 구현해야만 직렬화가 가능하다.



- 제어자 `transient`가 붙은 인스턴스변수는 직렬화 대상에서 제외된다.

```
public class UserInfo implements Serializable {  
    String name;  
    transient String password; // 직렬화 대상에서 제외된다.  
    int age;  
}
```

- `Serializable`을 구현하지 않은 클래스의 인스턴스도 직렬화 대상에서 제외

```
public class UserInfo implements Serializable {  
    String name;  
    transient String password;  
    int age;  
  
    Object obj = new Object(); // Object 객체는 직렬화할 수 없다.  
}
```

7.3 직렬화 가능한 클래스 만들기(2/2)

- **Serializable**을 구현하지 않은 조상의 멤버들은 직렬화 대상에서 제외된다.

```
public class SuperUserInfo {  
    String name;      // 직렬화되지 않는다.  
    String password; // 직렬화되지 않는다.  
}  
  
public class UserInfo extends SuperUserInfo implements Serializable {  
    int age;  
}
```

- **readObject()**와 **writeObject()**를 오버라이딩하면 직렬화를 마음대로...

```
private void writeObject(ObjectOutputStream out)  
    throws IOException {  
    out.writeUTF(name);  
    out.writeUTF(password);  
    out.defaultWriteObject();  
}  
  
private void readObject(ObjectInputStream in)  
    throws IOException, ClassNotFoundException {  
    name = in.readUTF();  
    password = in.readUTF();  
    in.defaultReadObject();  
}
```

7.4 직렬화 가능한 클래스의 버전 관리

- 직렬화했을 때와 역직렬화했을 때의 클래스가 같은지 확인할 필요가 있다.

```
java.io.InvalidClassException: UserInfo; local class incompatible: stream  
classdesc   serialVersionUID      =      6953673583338942489,    local    class  
serialVersionUID = -6256164443556992367  
...
```

- 직렬화할 때, 클래스의 버전(serialVersionUID)을 자동계산해서 저장한다.
- 클래스의 버전을 수동으로 관리하려면, 클래스 내에 정의해야 한다.

```
class MyData implements java.io.Serializable {  
    static final long serialVersionUID = 3518731767529258119L;  
    int value1;  
}
```

- **serialver.exe**는 클래스의 serialVersionUID를 자동생성해준다.

```
C:\jdk1.5\work\ch14>serialver MyData  
MyData: static final long serialVersionUID = 3518731767529258119L;
```

네트워킹(Networking)

1. 네트워킹(Networking)

1.1 클라이언트/서버(client/server)

1.2 IP주소(IP address)

1.3 InetAddress

1.4 URL(Uniform Resource Location)

1.5 URLConnection

2. 소켓 프로그래밍

2.1 TCP와 UDP

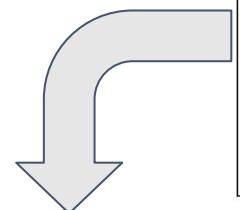
2.2 TCP소켓 프로그래밍

2.3 UDP소켓 프로그래밍

1. 네트워킹(Networking)

1.1 클라이언트/서버(client/server)

- 컴퓨터간의 관계를 역할(role)로 구분하는 개념
- 서비스를 제공하는 쪽이 서버, 제공받는 쪽이 클라이언트가 된다.
- 제공하는 서비스의 종류에 따라 메일서버(email server), 파일서버(file server), 웹서버(web server) 등이 있다.
- 전용서버를 두는 것을 ‘서버기반 모델’, 전용서버없이 각 클라이언트가 서버역할까지 동시에 수행하는 것을 ‘P2P 모델’



서버기반 모델(server-based model)	P2P 모델(peer-to-peer model)
<ul style="list-style-type: none">- 안정적인 서비스의 제공이 가능하다.- 공유 데이터의 관리와 보안이 용이하다.- 서버구축비용과 관리비용이 듦다.	<ul style="list-style-type: none">- 서버구축 및 운영비용을 절감할 수 있다.- 자원의 활용을 극대화 할 수 있다.- 자원의 관리가 어렵다.- 보안이 취약하다.

서버리스(serverless)

- 개발자가 서버를 관리할 필요 없이 애플리케이션을 빌드하고 실행할 수 있도록 하는 클라우드 네이티브 개발 모델
- 클라우드 제공업체가 서버 인프라에 대한 프로비저닝, 유지 관리, 스케일링 등의 일상적인 작업 처리
- 개발자는 배포를 위해 코드를 컨테이너에 패키징 작업만 처리
- 서버리스 애플리케이션은 배포되고 나면 필요에 따라 자동으로 스케일 업되거나 스케일 다운 가능

1.2 IP주소(IP address)

- 10진수 IP 주소 192.168.10.100는 32비트 숫자 11000000101010000000101001100100
- 이 숫자는 이해하기 어려울 수 있으므로 8자리 2진수의 네 부분으로 나누고 이 8비트 섹션을 옥텟이라 한다
- IP 주소는 11000000.10101000.00001010.01100100이 된다.

192	168	10	100
1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 1 0 0			
네트워크 주소		호스트 주소	

- 다른 일반적인 서브넷 마스크는 다음과 같습니다.

10진수	이진수
255.255.255.192	1111111.11111111.1111111.11000000
255.255.255.224	1111111.11111111.1111111.11100000

1.2 IP주소(IP address)

- 컴퓨터(host, 호스트)를 구별하는데 사용되는 고유한 주소값
 - 4 byte의 정수로 ‘a.b.c.d’와 같은 형식으로 표현.(a,b,c,d는 0~255의 정수)
 - IP주소는 네트워크주소와 호스트주소로 구성되어 있다.

192	168	10	100																										
1	1	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	0	0	1	1	0	0	1	0	0
네트워크 주소			호스트 주소																										

- 네트워크주소가 같은 두 호스트는 같은 네트워크에 존재한다.
 - IP주소와 서브넷마스크를 ‘&’연산하면 네트워크주소를 얻는다

서브넷 마스크(Subnet Mask)

1.3 InetAddress

- IP주소를 다루기 위한 클래스

메서드	설명
byte[] getAddress()	IP주소를 byte배열로 반환한다.
static InetAddress[] getAllByName(String host)	도메인명(host)에 지정된 모든 호스트의 IP주소를 배열에 담아 반환한다.
static InetAddress getByAddress(byte[] addr)	byte배열을 통해 IP주소를 얻는다.
static InetAddress getByName(String host)	도메인명(host)을 통해 IP주소를 얻는다.
String getCanonicalHostName()	FQDN(fully qualified domain name)을 반환한다.
String getHostAddress()	호스트의 IP주소를 반환한다.
String getHostName()	호스트의 이름을 반환한다.
static InetAddress getLocalHost()	지역호스트의 IP주소를 반환한다.
boolean isMulticastAddress()	IP주소가 멀티캐스트 주소인지 알려준다.
boolean isLoopbackAddress()	IP주소가 loopback 주소(127.0.0.1)인지 알려준다.

```
InetAddress ip = InetAddress.getByName ("www.naver.com");
```

```
getHostName () :www.naver.com  
getHostAddress () :222.122.84.200  
toString () :www.naver.com/222.122.84.200  
getAddress () :[-34, 122, 84, -56]  
getAddress ()+256 :222.122.84.200.
```

```
InetAddress ip = InetAddress.getLocalHost();
```

```
getHostName () :mycom  
getHostAddress () :192.168.10.100
```

```
InetAddress[] ipArr = InetAddress.getAllByName ("www.naver.com");
```

```
ipArr[0] :www.naver.com/222.122.84.200  
ipArr[1] :www.naver.com/222.122.84.250  
ipArr[2] :www.naver.com/61.247.208.6
```

1.4 URL(Uniform Resource Location)

- 인터넷에 존재하는 서버들의 자원에 접근할 수 있는 주소.

`http://www.javachobo.com:80/sample/hello.html?referer=javachobo#index1`

프로토콜 : 자원에 접근하기 위해 서버와 통신하는데 사용되는 통신규약(http)

호스트명 : 자원을 제공하는 서버의 이름(www.javachobo.com)

포트번호 : 통신에 사용되는 서버의 포트번호(80)

경로명 : 접근하려는 자원이 저장된 서버상의 위치(/sample/)

파일명 : 접근하려는 자원의 이름(hello.html)

쿼리(query) : URL에서 '?'이후의 부분(referer=javachobo)

참조(anchor) : URL에서 '#'이후의 부분(index1)

```
URL url = new URL("http://www.javachobo.com/sample/hello.html");
URL url = new URL("www.javachobo.com", "/sample/hello.html");
URL url = new URL("http", "www.javachobo.com", 80, "/sample/hello.html");
```

```
url.getAuthority():www.javachobo.com:80
url.getContent():sun.net.www.protocol.http.HttpURLConnection$HttpInputStream@c17164
url.getDefaultPort():80
url.getPort():80
url.getFile():/sample/hello.html?referer=javachobo
url.getHost():www.javachobo.com
url.getPath():/sample/hello.html
url.getProtocol():http
url.getQuery():referer=javachobo
url.getRef():index1
url.getUserInfo():null
url.toExternalForm():http://www.javachobo.com:80/sample/hello.html?referer=javachobo#index1
url.toURI():http://www.javachobo.com:80/sample/hello.html?referer=javachobo#index1
```

1.5 URLConnection(1/4)

- 어플리케이션과 URL간의 통신연결을 위한 추상클래스

메서드	설명
void addRequestProperty(String key, String value)	지정된 키와 값을 RequestProperty에 추가한다. 기존에 같은 키가 있어도 값을 덮어쓰지 않는다.
void connect()	URL에 지정된 자원에 대한 통신연결을 한다.
boolean getAllowUserInteraction()	UserInteraction의 허용여부를 반환한다.
int getConnectTimeout()	연결종료시간을 천분의 일초로 반환한다.
Object getContent()	content객체를 반환한다.
Object getContent(Class[] classes)	content객체를 반환한다.
String getContentEncoding()	content의 인코딩을 반환한다.
int getContentLength()	content의 크기를 반환한다.
String getContentType()	content의 type을 반환한다.
long getDate()	헤더(header)의 date필드의 값을 반환한다.
boolean getDefaultAllowUserInteraction()	defaultAllowUserInteraction의 값을 반환한다.
String getDefaultRequestProperty(String key)	RequestProperty에서 지정된 키의 디폴트 값을 얻는다.
boolean getDefaultUseCaches()	useCache의 디폴트 값을 얻는다.
boolean getDoInput()	doInput필드값을 얻는다.
boolean getDoOutput()	doOutput필드값을 얻는다.
long getExpiration()	자원(URL)의 만료일자를 얻는다.(천분의 일초단위)
FileNameMap getFile NameMap()	FileNameMap(mimetable)을 반환한다.
String getHeaderField(int n)	헤더의 n번째 필드를 읽어온다.
String getHeaderField(String name)	헤더에서 지정된 이름의 필드를 읽어온다.
long getHeaderFieldDate(String name, long Default)	지정된 필드의 값을 날짜값으로 변환하여 반환한다. 필드값이 유효하지 않을 경우 Default값을 반환한다.

1.5 URLConnection(2/4)

메서드	설명
int getHeaderFieldInt(String name,int Default)	지정된 필드의 값을 정수값으로 변환하여 반환한다. 필드값이 유효하지 않을 경우 Default값을 반환한다.
String getHeaderFieldKey(int n)	헤더의 n번째 필드를 읽어온다.
Map getHeaderFields()	헤더의 모든 필드와 값이 저장된 Map을 반환한다.
long getLastModifiedSince()	ifModifiedSince(변경여부)필드의 값을 반환한다.
InputStream getInputStream()	URLConnection에서 InputStream을 반환한다.
long getLastModified()	LastModified(최종변경일)필드의 값을 반환한다.
OutputStream getOutputStream()	URLConnection에서 OutputStream을 반환한다.
Permission getPermission()	Permission(허용권한)을 반환한다.
int getReadTimeout()	읽기제한시간의 값을 반환한다.(천분의 일초)
Map getRequestProperties()	RequestProperties에 저장된 (키, 값)을 Map으로 반환한다.
String getRequestProperty(String key)	RequestProperty에서 지정된 키의 값을 반환한다.
URL getURL()	URLConnection의 URL의 반환한다.
boolean getUseCaches()	캐쉬의 사용여부를 반환한다.
String guessContentTypeFromName(String fname)	지정된 파일(fname)의 content-type을 추측하여 반환한다.
String guessContentTypeFromStream(InputStream is)	지정된 입력스트림(is)의 content-type을 추측하여 반환한다.
void setAllowUserInteraction(boolean allowuserinteraction)	UserInteraction의 허용여부를 설정한다.
void setConnectTimeout(int timeout)	연결종료시간을 설정한다.
void setContentHandlerFactory(ContentHandlerFactory fac)	ContentHandlerFactory를 설정한다.
void setDefaultAllowUserInteraction(boolean defaultallowuserinteraction)	UserInteraction허용여부의 기본값을 설정한다.
void setDefaultRequestProperty(String key, String value)	RequestProperty의 기본 키쌍(key-pair)을 설정한다.
void setDefaultUseCaches(boolean defaultusecaches)	캐쉬 사용여부의 기본값을 설정한다.
void setDoInput(boolean doinput)	DoInput필드의 값을 설정한다.
void setDoOutput(boolean dooutput)	DoOutput필드의 값을 설정한다.
void setFileNameMap(FileNameMap map)	FileNameMap을 설정한다.

1.5 URLConnection(3/4)

메서드	설명
void setIfModifiedSince(long ifmodifiedsince)	ModifiedSince필드의 값을 설정한다.
void setReadTimeout(int timeout)	읽기제한시간을 설정한다.(천분의 일초)
void setRequestProperty(String key, String value)	RequestProperty에 (key, value)를 저장한다.
void setUseCaches(boolean usecaches)	캐쉬의 사용여부를 설정한다.

```
conn.toString():sun.net.www.protocol.http.HttpURLConnection:http://www.javachobo.com/sample/hello.html
getAllowUserInteraction():false
getConnectTimeout():0
getContent():sun.net.www.protocol.http.HttpURLConnection$HttpInputStream@61de33
getContentEncoding():null
getContentLength():174
getContentType():text/html
getDate():1189338850000
getDefaultValueAllowUserInteraction():false
getDefaultValueUseCaches():true
getDoInput():true
getDoOutput():false
getExpiration():0
getHeaderFields():{Content-Length=[174], Connection=[Keep-Alive], ETag=["12e391-ae-46dad401"], Date=[Sun, 09 Sep 2007 11:54:10 GMT], Keep-Alive=[timeout=5, max=60], Accept-Ranges=[bytes], Server=[RC-Web Server], Content-Type=[text/html], null=[HTTP/1.1 200 OK], Last-Modified=[Sun, 02 Sep 2007 15:17:21 GMT] }
getIfModifiedSince():0
getLastModified():1188746241000
getReadTimeout():0
getURL():http://www.javachobo.com/sample/hello.html
getUseCaches():true
```

1.5 URLConnection(4/4) - 예제

```
import java.net.*;
import java.io.*;

public class NetworkEx4 {
    public static void main(String args[]) {
        URL url = null;
        BufferedReader input = null;
        String address = "http://www.javachobo.com/sample/hello.html";
        String line = "";

        try {
            url = new URL(address);
            input = new BufferedReader(new InputStreamReader(url.openStream()));

            while((line=input.readLine()) !=null) {
                System.out.println(line);
            }
            input.close();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

InputStreamReader(InputStream in, String encoding)
지정된 인코딩을 사용하는 InputStreamReader를 생성한다.

URLConnection conn = url.openConnection();
InputStream in = conn.getInputStream();

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Sample Document</TITLE>
</HEAD>
<BODY>
Hello, everybody.
</BODY>
</HTML>
```

2. 소켓 프로그래밍

2.1 TCP와 UDP

▶ 소켓 프로그래밍이란?

- 소켓을 이용한 통신 프로그래밍을 뜻한다.
- 소켓(socket)이란, 프로세스간의 통신에 사용되는 양쪽 끝단(end point)
- 전화할 때 양쪽에 전화기가 필요한 것처럼, 프로세스간의 통신에서도 양쪽에 소켓이 필요하다.

▶ TCP와 UDP

- TCP/IP프로토콜에 포함된 프로토콜. OSI 7계층의 전송계층에 해당

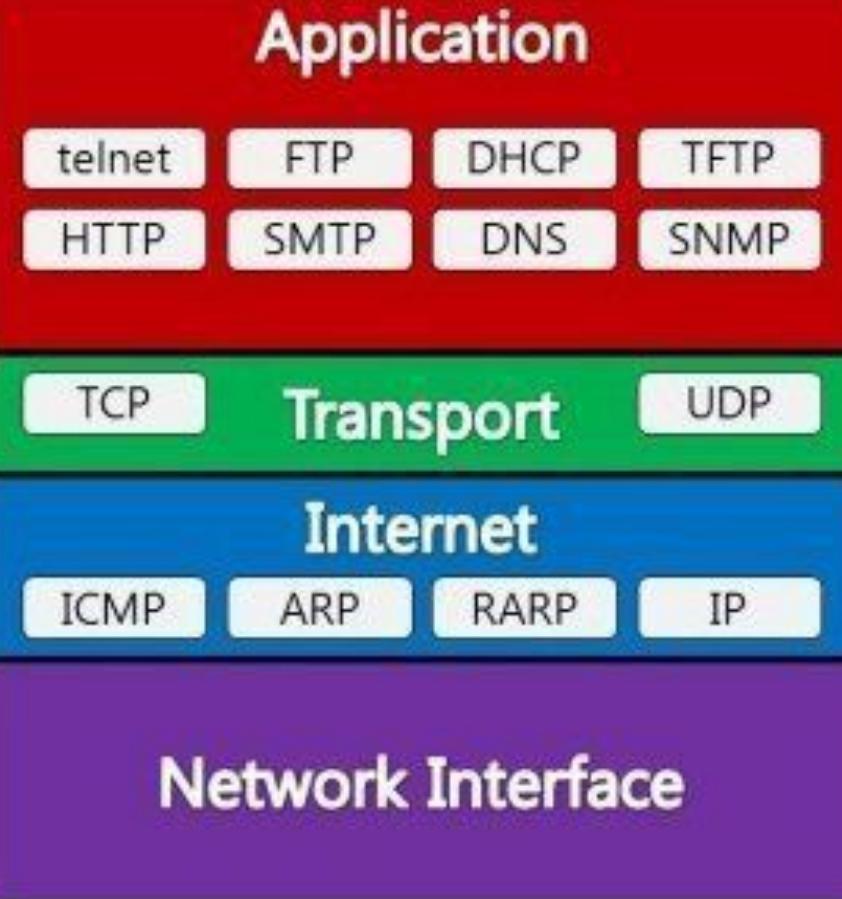
항목	TCP	UDP
연결방식	.연결기반(connection-oriented) - 연결 후 통신(전화기) - 1:1 통신방식	.비연결기반(connectionless-oriented) - 연결없이 통신(소포) - 1:1, 1:n, n:n 통신방식
특징	.데이터의 경계를 구분안함 (byte-stream) .신뢰성 있는 데이터 전송 - 데이터의 전송순서가 보장됨 - 데이터의 수신여부를 확인함 (데이터가 손실되면 재전송됨) - 패킷을 관리할 필요가 없음 .UDP보다 전송속도가 느림	.데이터의 경계를 구분함.(datagram) .신뢰성 없는 데이터 전송 - 데이터의 전송순서가 바뀔 수 있음 - 데이터의 수신여부를 확인안함 (데이터가 손실되어도 알 수 없음) - 패킷을 관리해주어야 함 .TCP보다 전송속도가 빠름
관련 클래스	.Socket .ServerSocket	.DatagramSocket .DatagramPacket .MulticastSocket

OSI 7 Layer Model

7 Layer
6 Layer
5 Layer
4 Layer
3 Layer
2 Layer
1 Layer



TCP/IP Protocol



OSI 7 Layer

OSI 모형(Open Systems Interconnection Reference Model)은 국제표준화기구(ISO)에서 개발한 모델로, 컴퓨터 네트워크 프로토콜 디자인과 통신을 계층으로 나누어 설명한 것이다. 일반적으로 OSI 7 계층이라고 한다.

목적

이 모델은 프로토콜을 기능별로 나눈 것이다. 각 계층은 하위 계층의 기능만을 이용하고, 상위 계층에게 기능을 제공한다. '프로토콜 스택' 혹은 '스택'은 이러한 계층들로 구성되는 프로토콜 시스템이 구현된 시스템을 가리키는데, 프로토콜 스택은 하드웨어나 소프트웨어 혹은 둘의 혼합으로 구현될 수 있다. 일반적으로 하위 계층들은 하드웨어로, 상위 계층들은 소프트웨어로 구현된다.

계층 1: 물리 계층

물리 계층(Physical layer)은 네트워크의 기본 네트워크 하드웨어 전송 기술을 이룬다. 네트워크의 높은 수준의 기능의 논리 데이터 구조를 기초로 하는 필수 계층이다. 다양한 특징의 하드웨어 기술이 접목되어 있기에 OSI 아키텍처에서 **가장 복잡한 계층**으로 간주된다.

계층 2: 데이터 링크 계층

데이터 링크 계층(Data[3]link layer)은 **포인트 투 포인트(Point to Point)** 간 신뢰성있는 전송을 보장하기 위한 계층으로 **CRC 기반의 오류 제어와 흐름 제어**가 필요하다. 네트워크 위의 개체들 간 데이터를 전달하고, 물리 계층에서 발생할 수 있는 오류를 찾아내고, 수정하는 데 필요한 기능적, 절차적 수단을 제공한다. 주소 값은 물리적으로 할당 받는데, 이는 네트워크 카드가 만들어질 때부터 맥 주소(MAC address)가 정해져 있다는 뜻이다. 주소 체계는 계층이 없는 단일 구조이다. 데이터 링크 계층의 가장 잘 알려진 예는 이더넷이다. 이 외에도 HDLC나 ADCCP 같은 포인트 투 포인트(point-to-point) 프로토콜이나 패킷 스위칭 네트워크나 LLC, ALOHA 같은 근거리 네트워크용 프로토콜이 있다. 네트워크 브릿지나 스위치 등이 이 계층에서 동작하며, 직접 이어진 곳에만 연결할 수 있다.

- 프레임에 주소부여(MAC - 물리적주소)
- 에러검출/재전송/흐름제어

계층 3: 네트워크 계층

네트워크 계층(Network layer)은 여러개의 노드를 거칠때마다 경로를 찾아주는 역할을 하는 계층으로 다양한 길이의 데이터를 네트워크들을 통해 전달하고, 그 과정에서 전송 계층이 요구하는 서비스 품질(QoS)을 제공하기 위한 기능적, 절차적 수단을 제공한다. 네트워크 계층은 라우팅, 흐름 제어, 세그멘테이션(segmentation/desegmentation), 오류 제어, 인터네트워킹(Internetworking) 등을 수행한다. 라우터가 이 계층에서 동작하고 이 계층에서 동작하는 스위치도 있다. 데이터를 연결하는 다른 네트워크를 통해 전달함으로써 인터넷이 가능하게 만드는 계층이다. 논리적인 주소 구조(IP), 곧 네트워크 관리자가 직접 주소를 할당하는 구조를 가지며, 계층적(hierarchical)이다.

서브네트의 최상위 계층으로 경로를 설정하고, 청구 정보를 관리한다. 개방형 시스템들의 사이에서 네트워크 연결을 설정, 유지, 해제하는 기능을 부여하고, 전송 계층 사이에 네트워크 서비스 데이터 유닛(NSDU : Network Service Data Unit)을 교환하는 기능을 제공한다.

- 주소부여(IP)
- 경로설정(Route)

계층 4: 전송 계층

전송 계층(Transport layer)은 양 끝단(End to end)의 사용자들이 신뢰성 있는 데이터를 주고 받을 수 있도록 해 주어, 상위 계층들이 데이터 전달의 유효성이나 효율성을 생각하지 않도록 해준다. 시퀀스 넘버 기반의 오류 제어 방식을 사용한다. 전송 계층은 특정 연결의 유효성을 제어하고, 일부 프로토콜은 상태 개념이 있고(stateful), 연결 기반(connection oriented)이다. 이는 전송 계층이 패킷들의 전송이 유효한지 확인하고 전송 실패한 패킷들을 다시 전송한다는 것을 뜻한다. 가장 잘 알려진 전송 계층의 예는 TCP이다.

종단간(end-to-end) 통신을 다루는 최하위 계층으로 종단간 신뢰성 있고 효율적인 데이터를 전송하며, 기능은 오류검출 및 복구와 흐름제어, 중복검사 등을 수행한다.

- 패킷 생성(Assembly/Sequencing/Deassembly/Error detection/Request repeat/Flow control)

계층 5: 세션 계층

세션 계층(Session layer)은 양 끝단의 응용 프로세스가 통신을 관리하기 위한 방법을 제공한다. 동시 송수신 방식(duplex), 반이중 방식(half-duplex), 전이중 방식(Full Duplex)의 통신과 함께, 체크 포인팅과 유휴, 종료, 다시 시작 과정 등을 수행한다. 이 계층은 TCP/IP 세션을 만들고 없애는 책임을 진다.

통신하는 사용자들을 동기화하고 오류복구 명령들을 일괄적으로 다룬다.

- 통신을 하기 위한 세션을 확립/유지/중단 (운영체제가 해줌)

계층 6: 표현 계층

표현 계층(Presentation layer)은 코드 간의 번역을 담당하여 사용자 시스템에서 데이터의 형식상 차이를 다루는 부담을 응용 계층으로부터 덜어 준다. MIME 인코딩이나 암호화 등의 동작이 이 계층에서 이루어진다. 예를 들면, EBCDIC로 인코딩된 문서 파일을 ASCII로 인코딩된 파일로 바꿔 주는 것이 표현 계층의 몫이다.

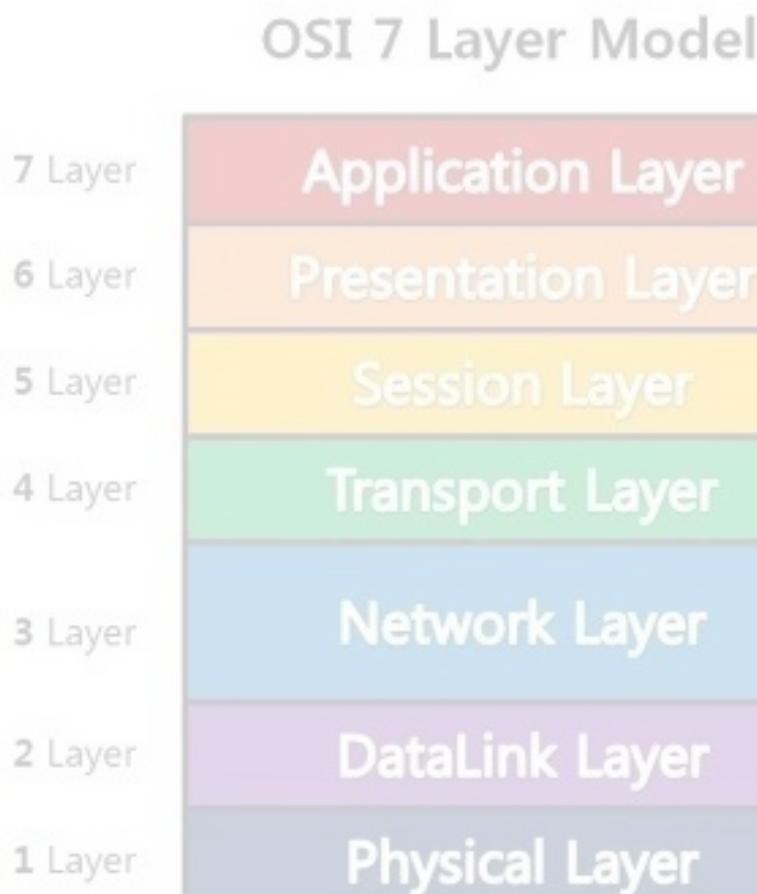
- 사용자의 명령어를 완성 및 결과 표현.
- 포장/압축/암호화

계층 7: 응용 계층

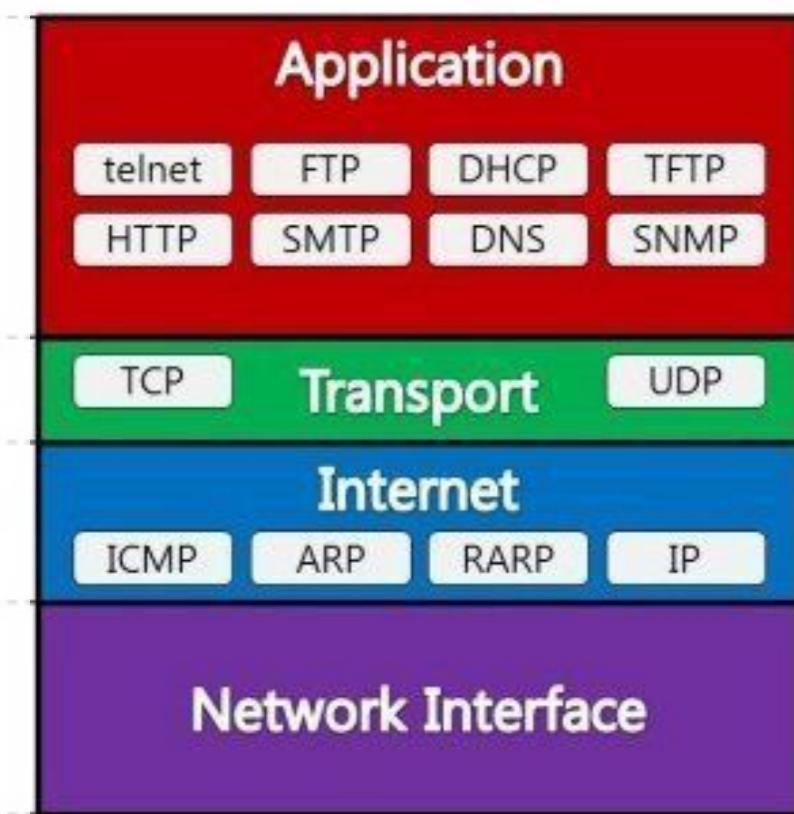
응용 계층(Application layer)은 응용 프로세스와 직접 관계하여 일반적인 응용 서비스를 수행한다. 일반적인 응용 서비스는 관련된 응용 프로세스들 사이의 전환을 제공한다. 응용 서비스의 예로, 가상 터미널(예를 들어, 텔넷), "Job transfer and Manipulation protocol" (JTM, 표준 ISO/IEC 8832) 등이 있다.

- 네트워크 소프트웨어 UI 부분
- 사용자의 입출력(I/O)부분

TCP/IP 4 Layer



TCP/IP Protocol



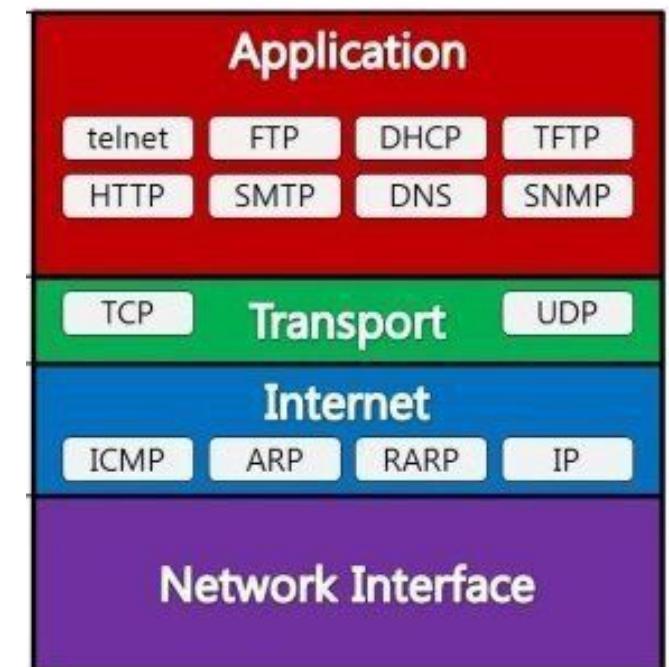
(1) Network Access Layer

- ① OSI 7 Layer에서 물리계층과 데이터링크 계층에 해당한다.
- ② OS의 네트워크 카드와 디바이스 드라이버 등과 같이 하드웨어적인 요소와 관련되는 모든 것을 지원하는 계층
- ③ 송신측 컴퓨터의 경우 상위 계층으로부터 전달받은 패킷에 물리적인 주소 MAC 주소 정보를 가지고 있는 헤더를 추가하여 프레임을 만들고, 프레임을 하위계층인 물리 계층으로 전달한다.
- ④ 수신측 컴퓨터의 경우 데이터 링크 계층에서 추가된 헤더를 제거하여 상위 계층인 네트워크 계층으로 전달한다.

(2) Internet Layer

- ① OSI 7 Layer의 네트워크 계층에 해당한다.
- ② 인터넷 계층의 주요 기능은 상위 트랜스포트 계층으로부터 받은 데이터에 IP패킷 헤더를 붙여 IP패킷을 만들고 이를 전송하는 것이다.

TCP/IP Protocol



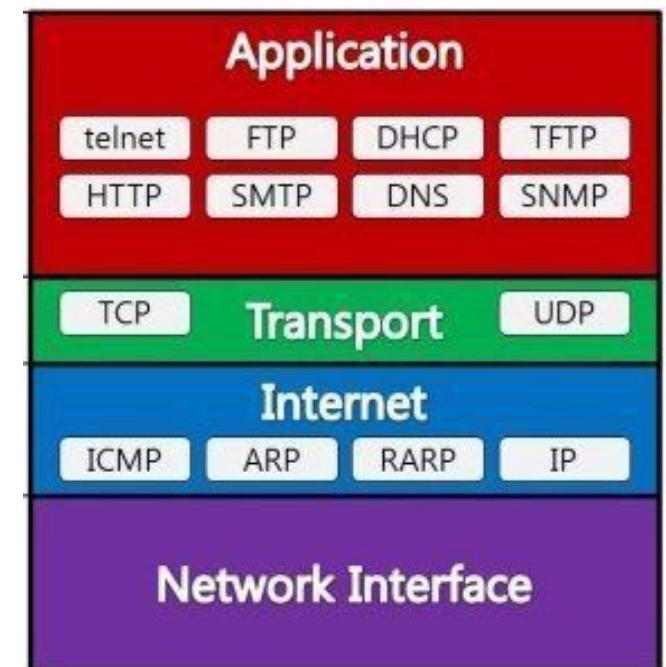
(3) Transport Layer

- ① OSI 7 Layer에서 전송계층에 해당한다.
- ② 네트워크 양단의 송수신 호스트 사이에서 신뢰성 있는 전송기능을 제공한다.
- ③ 시스템의 논리주소와 포트를 가지고 있어서 각 상위 계층의 프로세스를 연결해서 통신한다.
- ④ 정확한 패킷의 전송을 보장하는 TCP와 정확한 전송을 보장하지 않는 UDP 프로토콜을 이용한다.
- ⑤ 데이터의 정확한 전송보다 빠른 속도의 전송이 필요한 멀티미디어 통신에서 UDP를 사용하면 TCP보다 유용하다.

(4) Application Layer

- ① OSI 7 Layer에서 세션계층, 프레젠테이션계층, 애플리케이션 계층에 해당한다.
- ② 응용프로그램들이 네트워크서비스, 메일서비스, 웹서비스 등을 할 수 있도록 표준적인 인터페이스를 제공한다.

TCP/IP Protocol



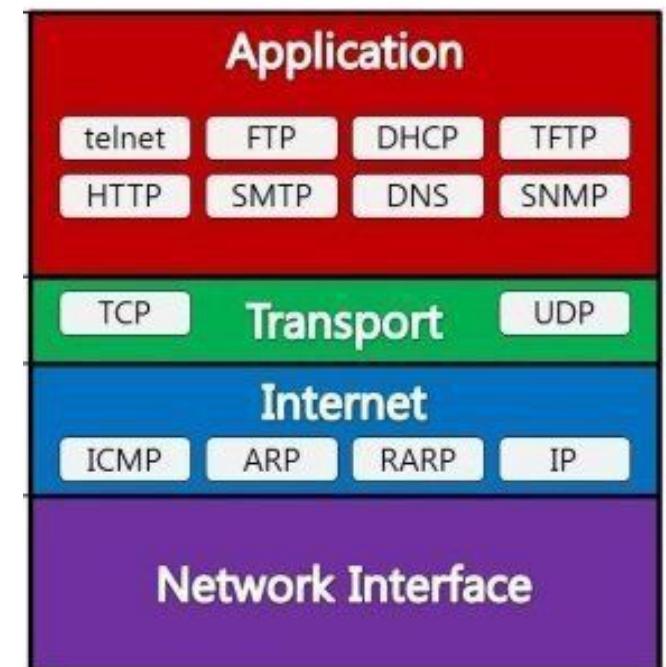
(3) Transport Layer

- ① OSI 7 Layer에서 전송계층에 해당한다.
- ② 네트워크 양단의 송수신 호스트 사이에서 신뢰성 있는 전송기능을 제공한다.
- ③ 시스템의 논리주소와 포트를 가지고 있어서 각 상위 계층의 프로세스를 연결해서 통신한다.
- ④ 정확한 패킷의 전송을 보장하는 TCP와 정확한 전송을 보장하지 않는 UDP 프로토콜을 이용한다.
- ⑤ 데이터의 정확한 전송보다 빠른 속도의 전송이 필요한 멀티미디어 통신에서 UDP를 사용하면 TCP보다 유용하다.

(4) Application Layer

- ① OSI 7 Layer에서 세션계층, 프레젠테이션계층, 애플리케이션 계층에 해당한다.
- ② 응용프로그램들이 네트워크서비스, 메일서비스, 웹서비스 등을 할 수 있도록 표준적인 인터페이스를 제공한다.

TCP/IP Protocol



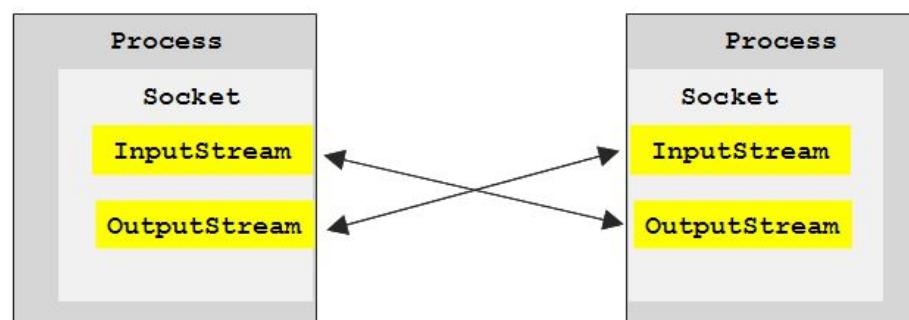
2.2 TCP소켓 프로그래밍

- 클라이언트와 서버간의 1:1 소켓 통신.
- 서버가 먼저 실행되어 클라이언트의 연결요청을 기다리고 있어야 한다.

1. 서버는 서버소켓을 사용해서 서버의 특정포트에서 클라이언트의 연결요청을 처리할 준비를 한다.
2. 클라이언트는 접속할 서버의 IP주소와 포트정보로 소켓을 생성해서 서버에 연결을 요청한다.
3. 서버소켓은 클라이언트의 연결요청을 받으면 서버에 새로운 소켓을 생성해서 클라이언트의 소켓과 연결되도록 한다.
4. 이제 클라이언트의 소켓과 새로 생성된 서버의 소켓은 서버소켓과 관계없이 1:1통신을 한다.

Socket - 프로세스간의 통신을 담당하며, InputStream과 OutputStream을 가지고 있다.
이 두 스트림을 통해 프로세스간의 통신(입출력)이 이루어진다.

ServerSocket - 포트와 연결(bind)되어 외부의 연결요청을 기다리다 연결요청이 들어오면,
Socket을 생성해서 소켓과 소켓간의 통신이 이루어지도록 한다.
한 포트에 하나의 ServerSocket만 연결할 수 있다.
(프로토콜이 다르면 같은 포트를 공유할 수 있다.)



2.2 TCP소켓 프로그래밍 - 예제

1. 서버프로그램을 실행한다.

```
> java.exe TcpIpServer
```

2. 서버소켓을 생성한다.

```
serverSocket = new ServerSocket(7777); // TcpIpServer.java
```

3. 서버소켓이 클라이언트 프로그램의 연결요청을 처리할 수 있도록 대기상태로 만든다. 클라이언트 프로그램의 연결요청이 오면 새로운 소켓을 생성해서 클라이언트 프로그램의 소켓과 연결한다.

```
Socket socket = serverSocket.accept(); // TcpIpServer.java
```

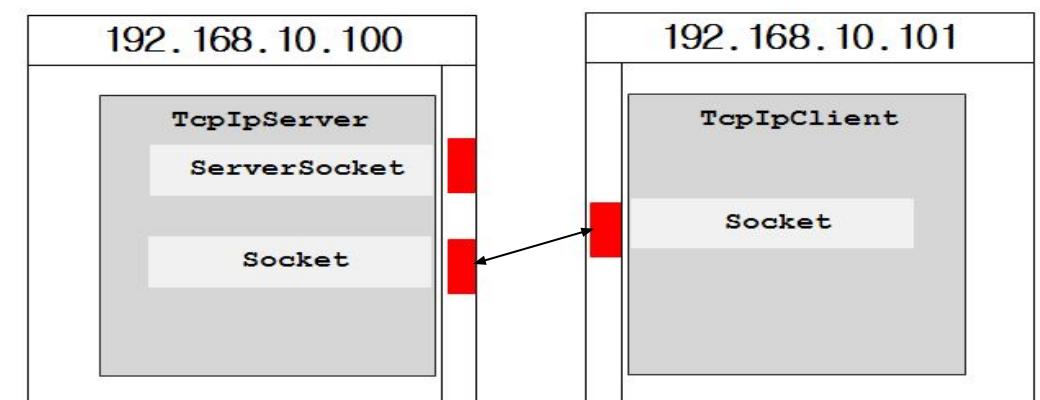
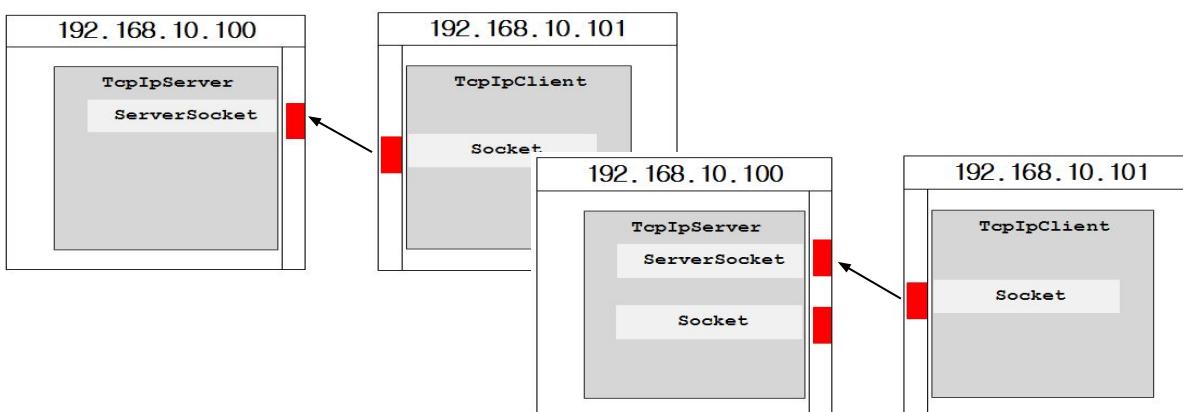
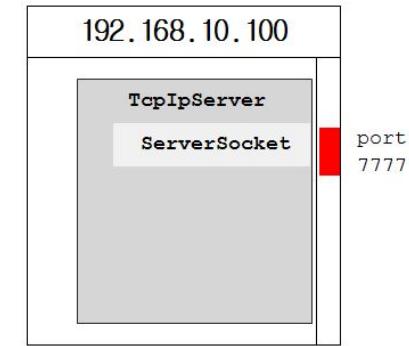
4. 클라이언트 프로그램(TcpIpClient.java)에서 소켓을 생성하여 서버소켓에 연결을 요청한다.

```
Socket socket = new Socket("192.168.10.100", 7777); // TcpIpClient.java
```

5. 서버소켓은 클라이언트 프로그램의 연결요청을 받아 새로운 소켓을 생성하여 클라이언트의 소켓과 연결한다.

```
Socket socket = serverSocket.accept(); // TcpIpServer.java
```

6. 새로 생성된 서버의 소켓(서버소켓 아님)은 클라이언트의 소켓과 통신한다.



2.3 UDP소켓 프로그래밍

- TCP소켓 프로그래밍에서는 **Socket**과 **ServerSocket**을 사용하지만,
UDP소켓 프로그래밍에서는 **DatagramSocket**과 **DatagramPacket**을 사용.
- UDP는 연결지향적이지 않으므로 연결요청을 받아줄 서버소켓이 필요없다.
- DatagramSocket간에 데이터(DatagramPacket)를 주고 받는다.

```
DatagramSocket socket = new DatagramSocket(7777);
DatagramPacket inPacket, outPacket;

byte[] inMsg = new byte[10];
byte[] outMsg;

while(true) {
    // 데이터를 수신하기 위한 패킷을 생성한다.
    inPacket = new DatagramPacket(inMsg, inMsg.length);

    // 패킷을 통해 데이터를 수신(receive)한다.
    socket.receive(inPacket);

    // 수신한 패킷으로부터 client의 IP주소와 Port를 얻는다.
    InetAddress address = inPacket.getAddress();
    int port = inPacket.getPort();
    ... 중간 생략...
    // 패킷을 생성해서 client에게 전송(send)한다.
    outPacket = new DatagramPacket(outMsg, outMsg.length, address, port);
    socket.send(outPacket);
}

DatagramSocket datagramSocket = new DatagramSocket();
InetAddress serverAddress = InetAddress.getByName("127.0.0.1");

byte[] msg = new byte[100]; // 데이터가 저장될 공간으로 byte배열을 생성한다.

DatagramPacket outPacket = new DatagramPacket(msg, 1, serverAddress, 7777);
DatagramPacket inPacket = new DatagramPacket(msg, msg.length);
datagramSocket.send(outPacket); // DatagramPacket을 전송한다.
datagramSocket.receive(inPacket); // DatagramPacket을 수신한다.
System.out.println("current server time :" + new String(inPacket.getData()));
```