

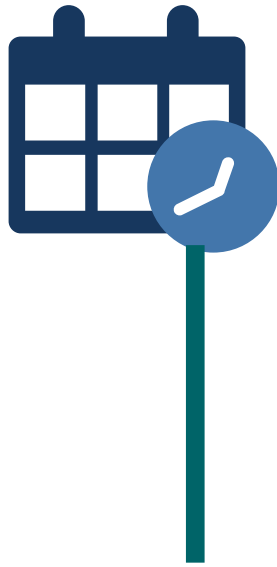
자바 기초 - 제네릭과 컬렉션

한성대학교 컴퓨터공학부

신 성



INDEX



- 제네릭 (Generic) 클래스
- Object 클래스 활용
- 멀티 타입 파라미터를 가진 제네릭 클래스
- 컬렉션
-

제네릭(Generic)

제네릭(Generic)

- 제네릭(Generic) 클래스 (Java 5 (Java 1.5)부터 추가)

☞ 컬렉션, 스트림, 람다식 그 외에 많은 API 등에서 널리 사용하고 있기 때문에 잘 이해해 두는 것이 필요

- 제네릭(Generic)은 ‘**일반화**’ 한다는 뜻을 담고 있음. 일반화 대상은 ‘**자료형**’

. 멤버 변수나 함수(매개 변수 또는 반환형)의 자료형(타입)을 사용자 마음대로 바꿀 수 있는 기능을

제공하는 클래스

```
class FruitBox <T> {  
    T item;  
    void set( T item ) {  
        this.item = item;  
    }  
    T get() {  
        return item;  
    }  
}
```

제네릭(Generic) 클래스는 이런 것!

☞ 자료형을 일반화

자료형 부분에는 int, double과 같은 기본 자료형은 사용 불가,
참조 자료형(클래스)만 사용 가능

기본 자료형을 사용하려면 Wrapper 클래스를 사용
(int를 사용하려면 Integer 클래스)

※ 제네릭 타입(자료형)

- 제네릭 타입에는 기본 자료형(타입)은 넣어줄 수 없고, 클래스 타입(클래스 명)만 넣어줄 수 있다.

`int a = 10;`

자료형

- ☞ 정수하나 저장할 수 있는 a라는 변수를 만들고,
이 정수하나 저장할 수 있는 a라는 변수에
10이라는 정수하나 저장해라.

`A a = new A();`

참조변수

객체에 접근할 수
있는 주소

- ☞ A 클래스의 객체를 저장할 수 있는 a라는 변수를 만들고,
이 A 클래스 객체를 저장할 수 있는 a라는 변수에
A 클래스 객체를 하나 만들어서 저장해라

제네릭(Generic)

- 제네릭(Generic) 클래스 (Java 5 (Java 1.5)부터 추가)

☞ 컬렉션, 스트림, 람다식 그 외에 많은 API 등에서 널리 사용하고 있기 때문에 잘 이해해 두는 것이 필요

- 제네릭(Generic)은 ‘일반화’ 한다는 뜻을 담고 있음. 일반화 대상은 ‘자료형’

. 멤버 변수나 함수(매개 변수 또는 반환형)의 자료형(타입)을 사용자 마음대로 바꿀 수 있는 기능을

제공하는 클래스

```
class FruitBox <T> {  
    T item;  
    void set( T item ) {  
        this.item = item;  
    }  
    T get() {  
        return item;  
    }  
}
```

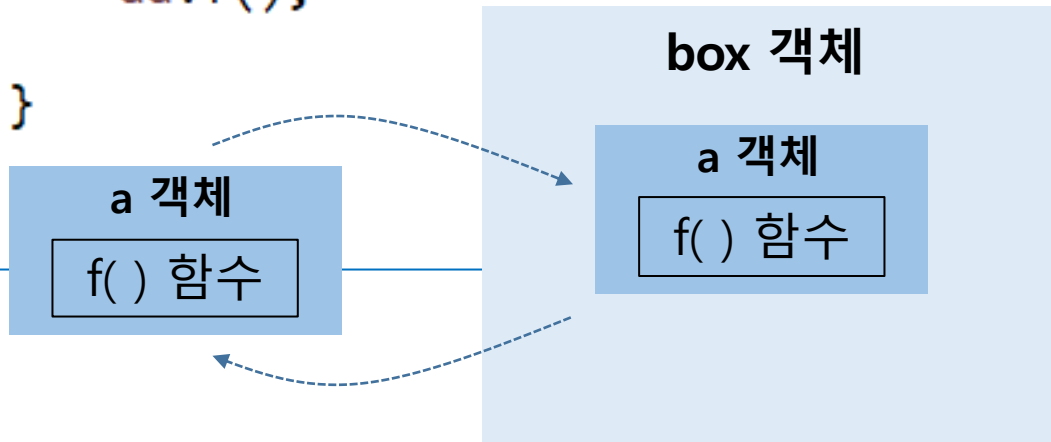
제네릭(Generic) 클래스는 이런 것!

☞ 자료형을 일반화

■ 제네릭 이해 목적의 예제

- 다른 객체를 저장할 용도의 간단한 Box 클래스 작성

```
public class Scratch {  
  
    public static void main(String[] args) {  
  
        A a = new A();  
        Box box = new Box();  
  
        box.set(a);  
        A aa = box.get();  
  
        aa.f();  
  
    }  
}
```



```
// 다른 객체를 전달받아서 저장할 용도의 클래스  
// (객체 저장용 클래스)
```

```
class Box{
```

```
    private A a;
```

```
    public void set(A a) {  
        this.a = a;  
    }
```

```
    public A get() {  
        return a;  
    }
```

```
}
```

문법상 모든 변수에 자료형(타입) 지정 필수
☞ 자료형 변수이름;

```
class A {  
    void f() {  
        System.out.println("f함수 호출");  
    }  
}
```

■ 실행 결과

```
public class Scratch {  
  
    public static void main(String[] args) {  
  
        A a = new A();  
        Box box = new Box();  
  
        box.set(a);  
        A aa = box.get();  
  
        aa.f();  
  
    }  
}
```

객체를 저장

실행 결과

Problems @ Javadoc Declaration Console ×
<terminated> Scratch (1) [Java Application] C:\WProgra
f함수 호출

```
// 다른 객체를 전달받아서 저장할 용도의 클래스  
// (객체 저장용 클래스)
```

```
class Box{
```

```
    private A a;
```

```
    public void set(A a) {  
        this.a = a;  
    }
```

```
    public A get() {  
        return a;  
    }
```

```
}
```

문법상 모든 변수에 자료형(타입) 지정 필수
☞ 자료형 변수이름;

```
class A {  
    void f() {  
        System.out.println("f함수 호출");  
    }  
}
```


- 자 이번에는 좀 더 구체적으로 만약 Apple 객체를 저장할 수 있는 AppleBox 클래스와 Orange 객체를 저장할 수 있는 OrangeBox 클래스가 필요하다고 한다면 다음과 같이 각각 작성해야 함

```
class AppleBox {  
    Apple item;  
  
    void set(Apple item) {  
        this.item = item;  
    }  
  
    Apple get() {  
        return item;  
    }  
}
```

```
class OrangeBox {  
    Orange item;  
  
    void set(Orange item) {  
        this.item = item;  
    }  
  
    Orange get() {  
        return item;  
    }  
}
```

객체의 종류 별로 여러 개의
클래스를 각각 작성해야 함

C언어 또는 Java 문법상 모든 변수에 자료형(타입) 지정 필수
☞ 자료형 변수이름;

※ [참고] (앞의 예제) 물론 다음과 같이 작성할 수도 있으나

클래스당 1개의 과일만 저장해야 하는 경우로 가정

```
class Box2 {  
  
    Apple a_item;  
    Orange o_item;  
  
    void set(Apple a_item, Orange o_item) {  
        this.a_item = a_item;  
        this.o_item = o_item;  
    }  
  
    Apple a_get() {  
        return a_item;  
    }  
  
    Orange o_get() {  
        return o_item;  
    }  
  
}
```

☞ 물론 이런 형태도

Apple과 Orange라는 자료형이 고정되는 것은 똑같고,

이런 형태도

제네릭의 타입 매개변수를 여러 개 넣어서
멀티 타입 제네릭 클래스로 만들 수가 있습니다.

- 이런 경우, 무엇이든 저장이 가능하도록 제네릭 클래스를 이용하여 자료형을 일반화하면 효율적 => 많이 사용

```
class AppleBox {  
    Apple item;  
  
    void set(Apple item) {  
        this.item = item;  
    }  
  
    Apple get() {  
        return item;  
    }  
}
```

```
class OrangeBox {  
    Orange item;  
  
    void set(Orange item) {  
        this.item = item;  
    }  
  
    Orange get() {  
        return item;  
    }  
}
```

타입 매개 변수 T

```
class FruitBox<T> {  
    T item;  
  
    void set(T item) {  
        this.item = item;  
    }  
  
    T get() {  
        return item;  
    }  
}
```

제네릭(Generic) 클래스

```
FruitBox<Apple> appleBox = new FruitBox<Apple>();  
FruitBox<Orange> orangeBox = new FruitBox<Orange>();
```

■ 제네릭 클래스 활용

```
class FruitBox<T> {  
    T item;  
  
    void set( T item ) {  
        this.item = item;  
    }  
  
    T get() {  
        return item;  
    }  
}
```

무엇이든(어떤 객체든)
저장 가능한 클래스

```
public class Scratch {
```

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

```
        Apple apple = new Apple();  
        FruitBox<Apple> appleBox = new FruitBox<Apple>();
```

☞ apple 객체를 저장할 수 있는 appleBox 객체 생성 및 활용

```
        appleBox.set(apple);  
        Apple apple2 = appleBox.get();  
        apple2.f();
```

```
        Orange orange = new Orange();  
        FruitBox<Orange> orangeBox = new FruitBox<Orange>();
```

☞ orange 객체를 저장할 수 있는 orangeBox 객체 생성 및 활용

```
        orangeBox.set(orange);  
        Orange orange2 = orangeBox.get();  
        orange2.f();
```

```
    }
```

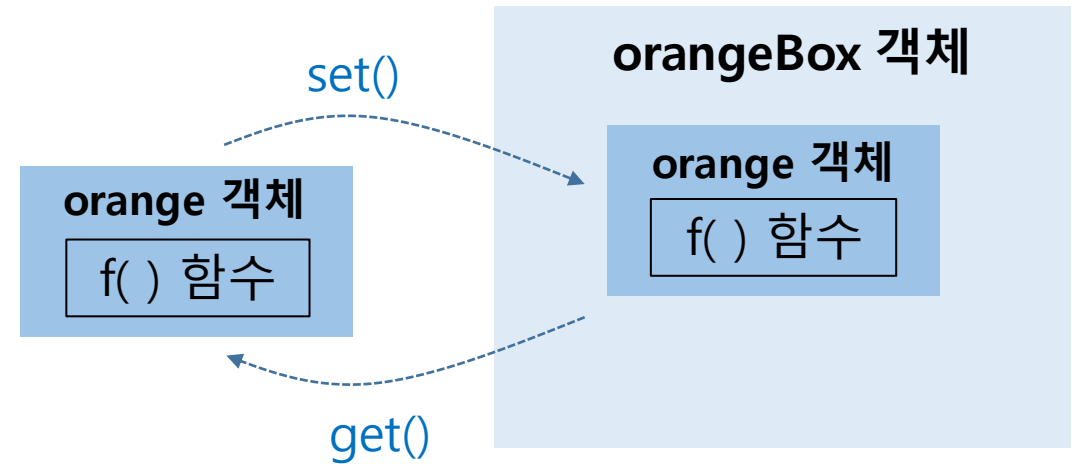
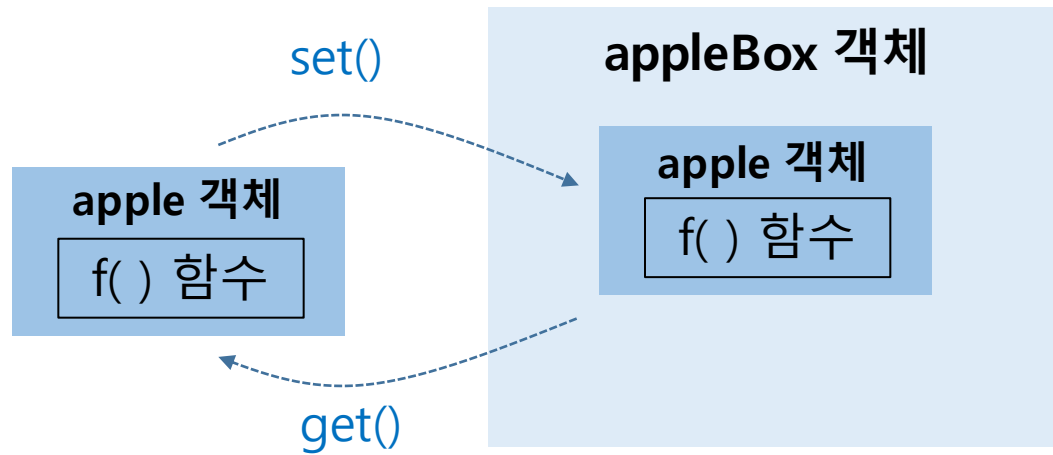
```
}
```

실행 결과 : Apple 객체의 f() 함수 호출
Orange 객체의 f() 함수 호출

```
class Apple {  
    void f() {  
        System.out.println("Apple 객체의 f() 함수 호출");  
    }  
}
```

```
class Orange{  
    void f() {  
        System.out.println("Orange 객체의 f() 함수 호출");  
    }  
}
```

- (앞의) 제네릭 클래스 활용 구조



Object 클래스 활용

- 물론 제네릭을 사용안하고 다음과 같이 Object 클래스를 활용하는 방법도 있지만 (예 : JSP 세션 내장 객체의 세터/게터 함수)
강제형변환 등을 해야해서 번거롭다. 또한 제네릭은 자료형의 불일치로 인한 실수 등 안정성 측면에서도 도움

☞ '업캐스팅/다운캐스팅'은
앞서 올려드린 강의 참고

```
class FruitBox2 {  
  
    Object item;  
  
    void set(Object item) {  
        this.item = item;  
    }  
  
    Object get() {  
        return item;  
    }  
  
}
```

모든 클래스의 최상위 클래스인 Object 클래스 활용

모든 클래스는 Object 클래스를 상속받으므로 어떤 객체든 업캐스팅을 통해 저장 가능
하지만 나중에 다시 꺼내서 쓰려고 할 때 반환형이 Object 이므로 강제 형변환 필요

[참고] (지난 학기에 배웠던) JSP '세션 내장 객체'의 경우

- setAttribute(), getAttribute() 메소드

```
void setAttribute(String name, Object value)
```

```
Object getAttribute(String name)
```

```
String s1 = "Apple";
```

☞ '업캐스팅/다운캐스팅'은 앞서 올려드린 강의 참고

```
session.setAttribute("add", s1);    //Object value = s1;  
                                   (String key, Object value)  Object 클래스 타입의 참조변수
```

```
String s2 = (String) session.getAttribute("add");
```

강제 형변환

반환형이 Object

- Object 클래스를 활용해 유사한 동작 구현 ➡ 무엇이든(어떤 객체든) 저장 가능한 클래스

```
class FruitBox2 {  
  
    Object item;  
  
    void set(Object item) {  
        this.item = item;  
    }  
  
    Object get() {  
        return item;  
    }  
  
}
```

```
public class Scratch {  
  
    public static void main(String[] args) {  
  
        Apple apple = new Apple();  
        FruitBox2 appleBox = new FruitBox2();  
        appleBox.set(apple);    //업캐스팅  
  
        Apple apple2 = (Apple) appleBox.get();  
        apple2.f();    // 반환되는 객체는 Object 클래스  
                        타입이므로 강제 형변환 필요  
  
        Orange orange = new Orange();  
        FruitBox2 orangeBox = new FruitBox2();  
        orangeBox.set(orange);    //업캐스팅  
  
        Orange orange2 = (Orange) orangeBox.get();  
        orange2.f();    // 반환되는 객체는 Object 클래스  
                        타입이므로 강제 형변환 필요  
  
    }  
}
```

멀티 타입 파라미터를 가진 제네릭 클래스

- 멀티 타입 파라미터를 가진 제네릭 클래스 (앞의 참고 부분 코드를 변경)

```
class Box2 {  
  
    Apple a_item;  
    Orange o_item;  
  
    void set(Apple a_item, Orange o_item) {  
        this.a_item = a_item;  
        this.o_item = o_item;  
    }  
  
    Apple a_get() {  
        return a_item;  
    }  
  
    Orange o_get() {  
        return o_item;  
    }  
  
}
```



```
class FruitBox3<T, M> {  
  
    T a_item;  
    M o_item;  
  
    void set(T a_item, M o_item) {  
        this.a_item = a_item;  
        this.o_item = o_item;  
    }  
  
    T a_get() {  
        return a_item;  
    }  
  
    M o_get() {  
        return o_item;  
    }  
  
}
```

멀티 타입 파라미터를 가진 제네릭 클래스

- 멀티 타입 파라미터를 가진 제네릭 클래스 활용

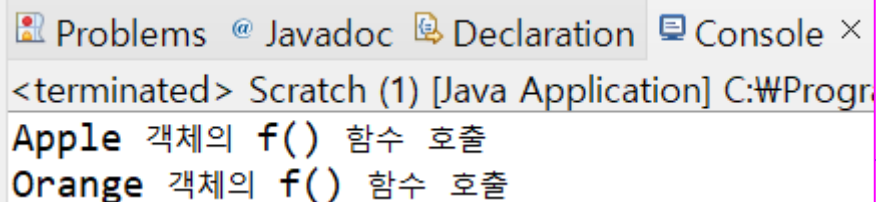
```
class FruitBox3<T, M> {  
  
    T a_item;  
    M o_item;  
  
    void set(T a_item, M o_item) {  
        this.a_item = a_item;  
        this.o_item = o_item;  
    }  
  
    T a_get() {  
        return a_item;  
    }  
  
    M o_get() {  
        return o_item;  
    }  
}
```

이클립스에서 자바 실습하는 부분은

이전 강의 (업캐스팅/다운캐스팅 PART2_실습) 부분 참고

```
class Scratch {  
  
    public static void main(String[] args) {  
  
        Apple apple = new Apple();  
        Orange orange = new Orange();  
        FruitBox3<Apple, Orange> appleBox = new FruitBox3<Apple, Orange>();  
  
        appleBox.set(apple, orange);  
  
        Apple apple2 = appleBox.a_get();  
        Orange orange2 = appleBox.o_get();  
  
        apple2.f();  
        orange2.f();  
  
    }  
}
```

실행 결과 :



The screenshot shows the Eclipse IDE's console window. At the top, there are tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active, displaying the output of the 'Scratch (1) [Java Application]' program. The output consists of two lines: 'Apple 객체의 f() 함수 호출' (Apple object's f() method call) and 'Orange 객체의 f() 함수 호출' (Orange object's f() method call).

컬렉션(collection)

컬렉션(collection)

- 컬렉션(collection) - ArrayList<E>에 대해 학습 ※ E는 의미 없음, 다른 문자 사용 가능

☞ 가변 길이 배열을 구현해 놓은 제네릭 클래스(제네릭 기법으로 구현)

- 컬렉션은 현대의 자바 프로그램을 작성하는데 빼놓을 수 없는 중요한 클래스
- 어플리케이션을 개발하다가 보면 다수의 객체를 저장해 두고 필요할 때마다 꺼내서 사용하는 경우가 많음
 - 가장 간단한 방법은 배열을 사용하는 것
 - 배열은 쉽게 생성하고 사용할 수 있지만, 저장할 수 있는 객체의 수가 배열을 생성할 때 결정(고정)
 - ArrayList<E>는 스스로 배열의 크기를 자동 조절하기 때문에 배열의 크기에 대해 신경 쓸 필요가 없음
 - ☞ 저장되는 객체의 수에 따라 자동적으로 용량이 늘어나는 가변 길이 배열(크기가 무한히 늘어나는 배열)
 - 쉽고 편하게 사용할 수 있는 다양한 메소드를 제공
 - ※ 컬렉션 클래스는 데이터를 저장하고 관리하는 클래스의 총칭, ArrayList<E>와 같은 가변 길이의 배열뿐만 아니라 연결 리스트, 해시 테이블, 트리 등 다양한 데이터 구조를 지원

ArrayList<E>

- ArrayList<E> 클래스 생성

- <E>에 요소로 사용할 특정 타입으로 구체화(모든 클래스의 객체 저장 가능)

```
ArrayList<String> a = new ArrayList<String>();
```

- 다음과 같이 초기 용량을 설정 가능하고, 만약 생략하면 10으로 설정

(지금의 경우 문자열 객체 5개 저장 가능, 이후 더 많은 수의 객체를 저장해도 자동적으로 용량 증가)

```
ArrayList<String> a = new ArrayList<String>(5);
```

ArrayList<E>

- ArrayList<E> 클래스 생성

- <E>에 요소로 사용할 특정 타입으로 구체화(모든 클래스의 객체 저장 가능)

```
ArrayList<String> a = new ArrayList<String>();
```

- 단, 기본 자료형은 불가능(제너릭 클래스로 구현되어 있으므로)

```
ArrayList<int> a = new ArrayList<int>();
```

- 기본 자료형은 Wrapper 클래스를 이용해 객체로 만든 후 저장

```
ArrayList<Integer> a = new ArrayList<Integer>();
```


[참고] String 클래스

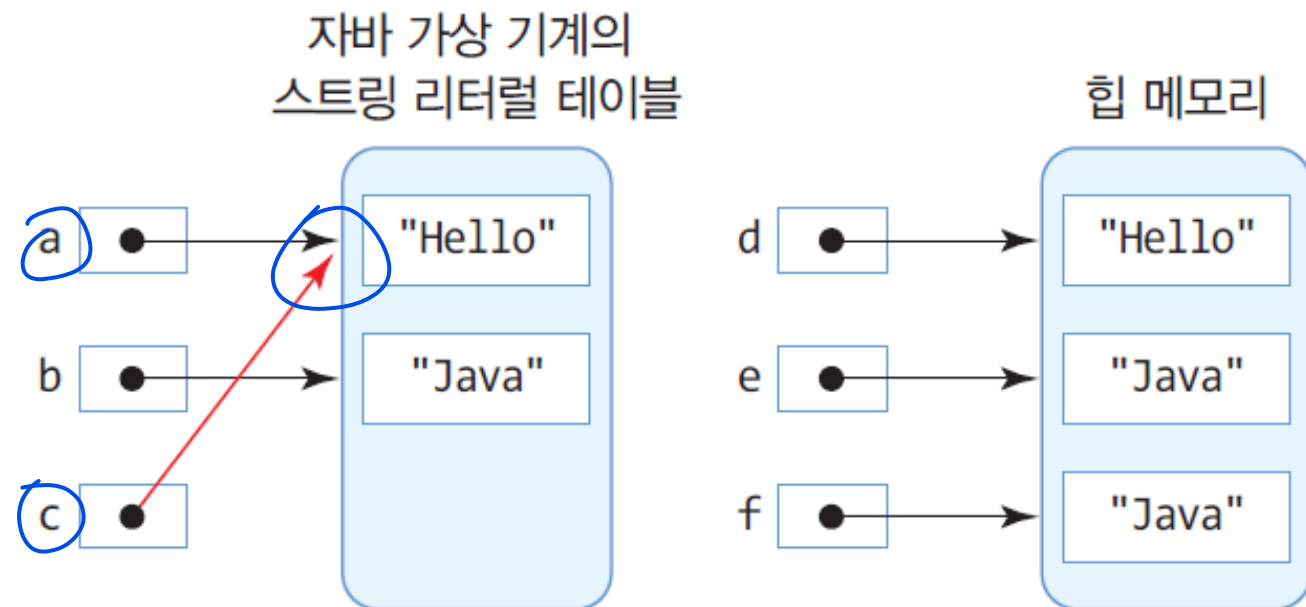
■ String 클래스

```
String s = new String("Hello");
```

```
String s = "Hello";
```

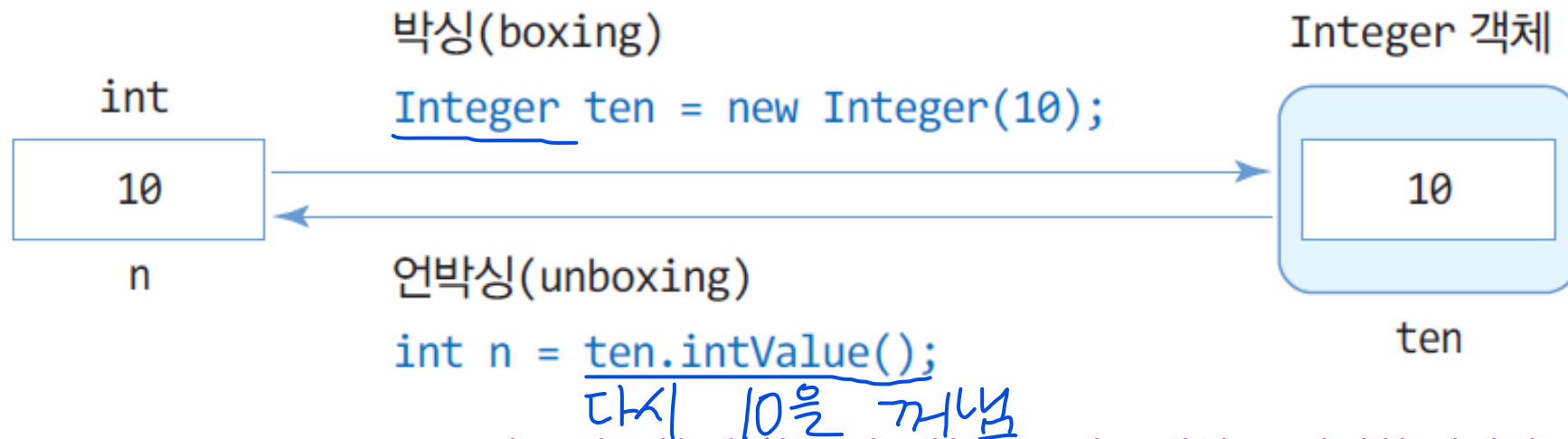
리터럴 상수로 취급

```
String a = "Hello"; a  
String b = "Java";  
String c = "Hello"; c  
String d = new String("Hello");  
String e = new String("Java");  
String f = new String("Java");
```



박싱과 언박싱

- 박싱(boxing) : 기본 타입(자료형)의 값을 Wrapper 객체로 변환하는 것
- 언박싱(unboxing): Wrapper 객체에 들어 있는 기본 타입(자료형)의 값을 빼내는 것, 박싱의 반대



■ 자동 박싱과 자동 언박싱

- JDK 5.0부터 박싱과 언박싱은 자동으로 이루어지도록 컴파일됨

```
Integer ten = 10;           // 자동 박싱. Integer ten = new Integer(10);과 동일
int n = ten;                // 자동 언박싱. int n = ten.intValue();와 동일
```

ArrayList<E>

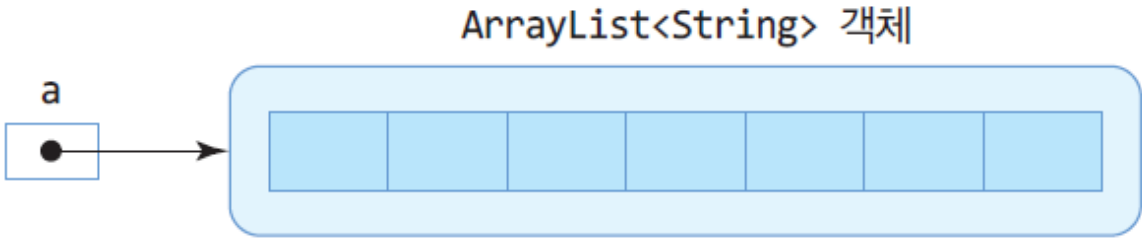
ArrayList<E> 클래스 안의 메소드

메소드	설명
boolean add(E element)	ArrayList의 맨 뒤에 element 추가
void add(int index, E element)	인덱스 index에 지정된 element 삽입
boolean addAll(Collection<? extends E> c)	컬렉션 c의 모든 요소를 ArrayList의 맨 뒤에 추가
void clear()	ArrayList의 모든 요소 삭제
boolean contains(Object o)	ArrayList가 지정된 객체를 포함하고 있으면 true 리턴
E elementAt(int index)	index 인덱스의 요소 리턴
E get(int index)	index 인덱스의 요소 리턴
int indexOf(Object o)	o와 같은 첫 번째 요소의 인덱스 리턴. 없으면 -1 리턴
boolean isEmpty()	ArrayList가 비어 있으면 true 리턴
E remove(int index)	index 인덱스의 요소 삭제
boolean remove(Object o)	o와 같은 첫 번째 요소를 ArrayList에서 삭제
int size()	ArrayList가 포함하는 요소의 개수 리턴
Object[] toArray()	ArrayList의 모든 요소를 포함하는 배열 리턴

ArrayList<E> 활용 사례

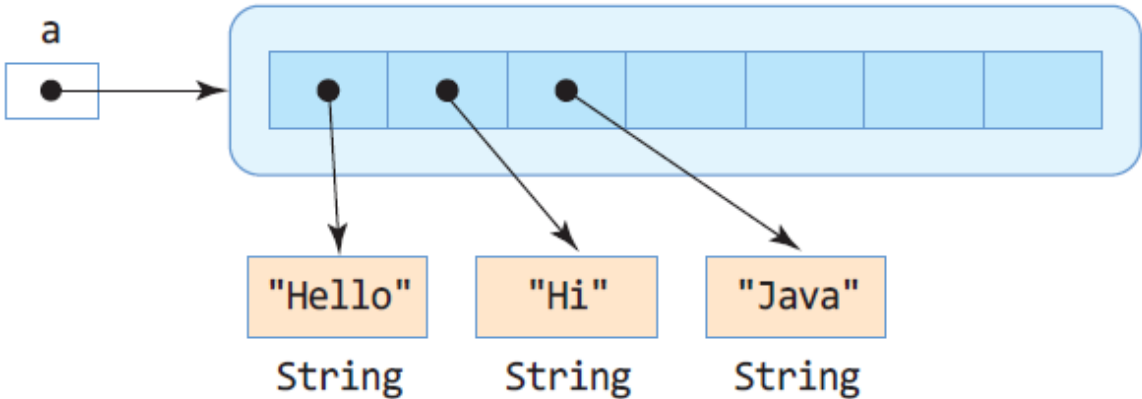
ArrayList 생성

```
ArrayList<String> a = new ArrayList<String>(7);
```



요소 삽입

```
a.add("Hello");  
a.add("Hi");  
a.add("Java");
```



요소 개수 n
벡터의 용량 c

```
int n = a.size(); // n은 3  
int c = a.capacity(); // capacity() 메소드 없음
```

n = 3

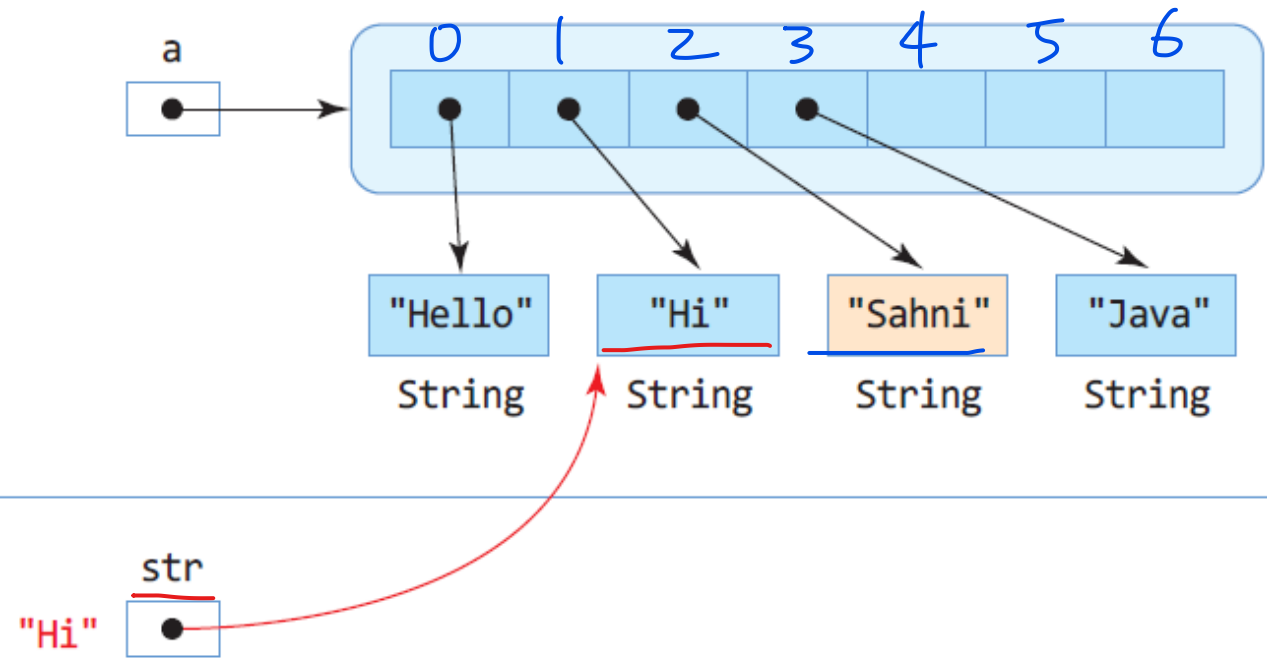
오류

ArrayList<E> 활용 사례

요소 중간 삽입 `a.add(2, "Sahni");`

오류 ~~`a.add(5, "Sahni");`~~
 // a.size()보다 큰 위치에 삽입 불가능, 오류

요소 알아내기 `String str = a.get(1);`



ArrayList<E> 활용 사례

요소 삭제

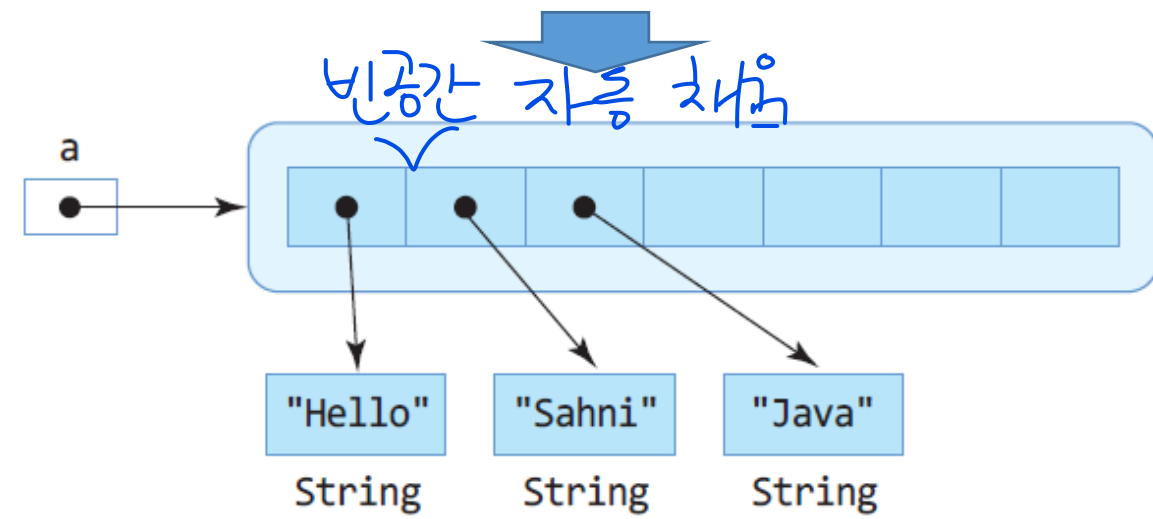
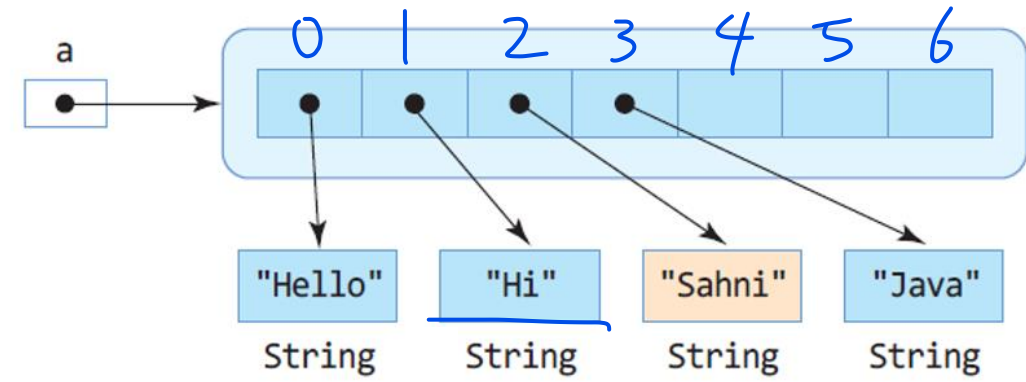
```
a.remove(1);
```

오류

```
a.remove(4); // 오류
```

모든 요소 삭제

```
a.clear(); 다 지우기
```



Q/A

☞ 궁금한 내용 등은

카톡, 문자, 메일, 전화, 방문, 이클래스 Q/A 등으로
언제든 편하게 물어보시기 바랍니다.

- 휴대폰 : 010-8873-8353
- 카카오톡 : sihns929 (또는 전화번호로 친구 등록)
- 메일 : sihns@hansung.ac.kr, 연구실: 우촌관 702호



 **T h a n k y o u**

TECHNOLOGY

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Velit ex
plicabo ipsum, labore sed tempora ratione asperiores des
cenderat bore sed tempora rati jgert one bore sed tem!