

Java Basic APIs

Language API

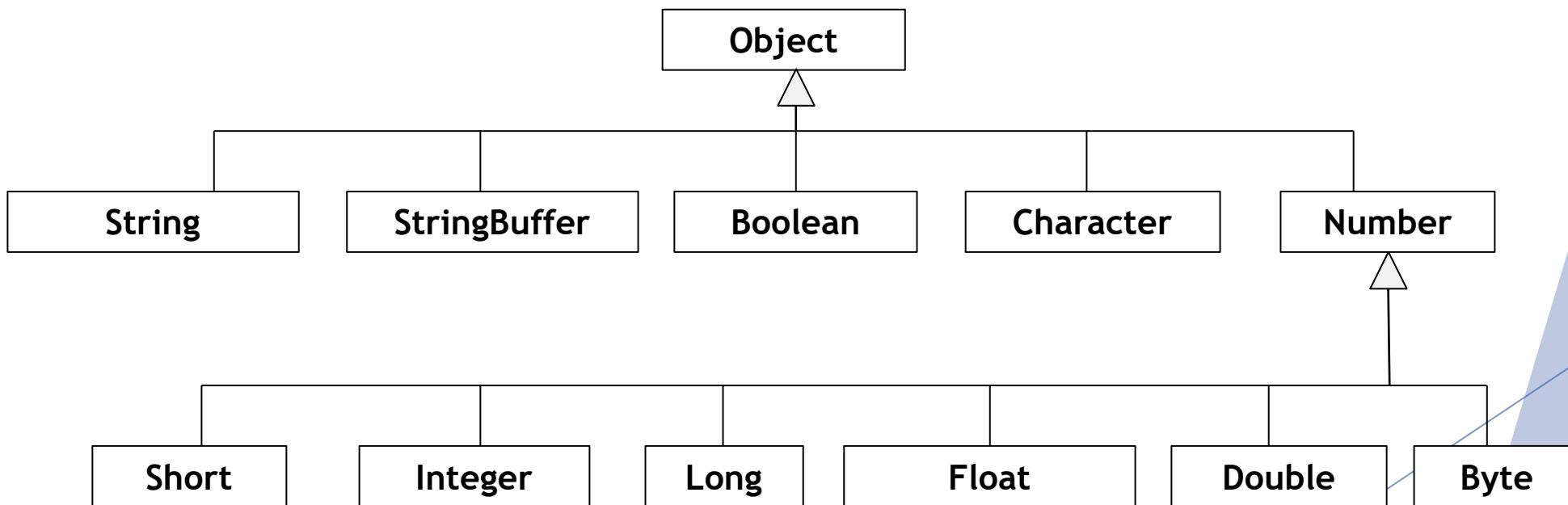
Java Basic APIs

Object Class

Language API

: java.lang

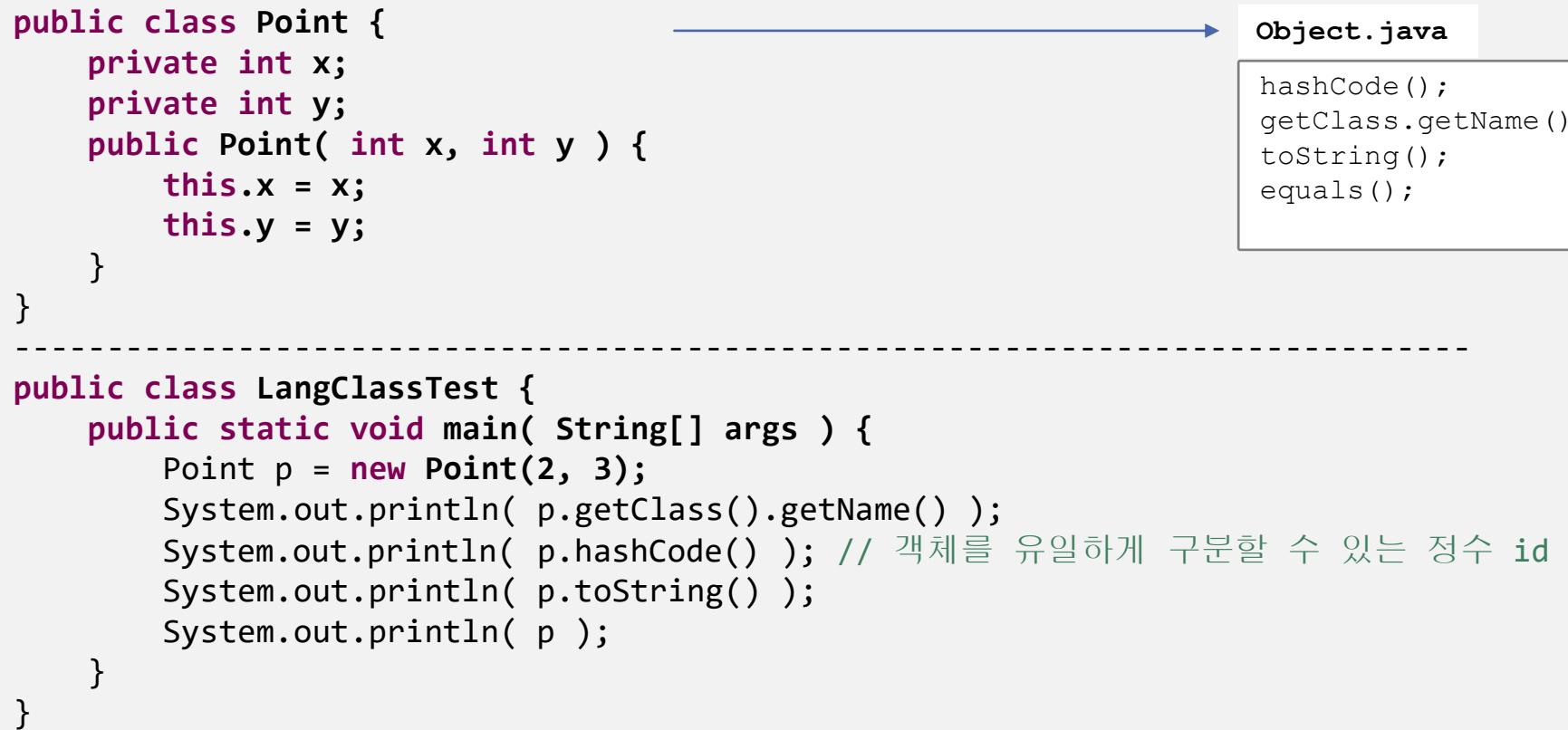
- ▶ 자바 프로그램에서 가장 많이 사용하는 패키지
- ▶ import 하지 않아도 자동으로 포함됨



Language API

: Object 클래스

- ▶ 모든 클래스의 최상위 클래스
- ▶ 명시적으로 `extends java.lang.Object` 하지 않아도 자동으로 상속받게 된다



기존에 만들어 둔 Point 클래스의
toString 메서드를 오버라이드 해 봅시다

Language API

: Object 클래스 toString() 메서드 오버라이드

- ▶ `toString()` : 객체의 정보를 문자열로 반환
- ▶ Object의 `toString` :
 - ▶ `getClass().getName() + "@" + Integer.toHexString(hashCode())`

`Object.java`

```
hashCode();  
getClass.getName();  
toString();  
equals();
```

```
public class Point {  
    private int x;  
    private int y;  
    public Point( int x, int y ) {  
        this.x = x;  
        this.y = y;  
    }  
    public String toString() {  
        return "Point(" + x + "," + y + ")"; // Point 클래스만의 toString()  
    }  
}
```



Language API

: equals()

- ▶ 두 객체의 비교시 == 와 Object 클래스의equals() 메서드를 사용한다
- ▶ == 와 equals()의 차이 : **확실히 구분**하여 사용
 - ▶ == 참조변수값 비교
 - ▶ equals() : 정의한 값 비교
- ▶ 참조 변수값을 먼저 비교한다
- ▶ 참조변수값이 같으면 두 객체는 같은 것으로 한다
- ▶ 참조변수값이 다르면 두 객체의 속성값을 비교한다

Language API

: 객체 비교 == vs equals() 메서드

```
public class LangClassTest {  
    public static void main(String[] args) {  
        Point a = new Point(2, 3);  
        Point b = new Point(2, 3);  
  
        System.out.println( a == b );  
        System.out.println( a.equals( b ) );  
  
        String s1 = new String("hello");  
        String s2 = new String("hello");  
  
        System.out.println( s1 == s2 );  
        System.out.println( s1.equals( s2 ) );  
    }  
}
```

Language API

: equals() 오버라이딩

```
public class Point {  
    private int x;  
    private int y;  
    public Point( int x, int y ) {  
        this.x = x;  
        this.y = y;  
    }  
    public boolean equals( Point p ) {  
        if( x == p.x && y == p.y) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}  
  
-----  
public class LangClassTest {  
    public static void main( String[] args ) {  
        Point a = new Point(2, 3);  
        Point b = new Point(2, 3);  
        System.out.println( a == b );  
        System.out.println( a.equals( b ) );  
    }  
}
```



실습예제

[문제]

- int 타입의 x, y, radius 필드를 가지는 Circle 클래스를 작성하세요.
- equals() 메소드를 재정의하여 두 개의 Circle 객체의 반지름이 같으면 두 Circle 객체가 동일한 것으로 판별하도록 하세요.

CircleApp.java

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

        Circle a = new Circle(6, 4, 10);
        Circle b = new Circle(2, 12, 10);
        Circle c = new Circle(3, 3, 12);
        Circle d = c;

        System.out.println(a.equals(b));
        System.out.println(a.equals(c));
        System.out.println(a.equals(d));
        System.out.println(d.equals(c));
    }
}
```

실습예제

[문제]

- Rectangle 클래스를 만들고 equals() 사용해 봅시다.
- int 타입의 width, height의 필드를 가지는 Rectangle 클래스를 작성하고 인스턴스를 생성합니다.
- 구해지는 면적이 같으면 두 객체가 같은 것으로 판별하도록 equals()를 오버라이딩 하세요.

RectangleApp.java

```
public class RectangleApp{  
    public static void main(String[] args){  
  
        Rectangle a = new Rectangle(6, 4);  
        Rectangle b = new Rectangle(2, 12);  
        Rectangle c = new Rectangle(3, 3);  
        Rectangle d = c;  
  
        System.out.println(a.equals(b));  
        System.out.println(a.equals(c));  
        System.out.println(a.equals(d));  
        System.out.println(d.equals(c));  
    }  
}
```

Language API

: 객체의 복제 - 얇은 복제

- ▶ 객체 복제 : 원본 객체의 값과 동일한 값을 가진 새로운 객체를 생성하는 것
- ▶ 얇은 복제 : 단순히 필드 값을 복사하는 방식으로 객체를 복제하는 것
- ▶ **Cloneable** 인터페이스를 구현하여 **clone** 메서드를 사용 가능하게 해야 한다

```
public class Point implements Cloneable {  
    private int x;  
    private int y;  
  
    // ...  
    public Point getClone() {  
        Point clone = null;  
        try {  
            clone = (Point)clone();  
        } catch (CloneNotSupportedException e) {}  
        return clone;  
    }  
}
```

Language API

: 객체의 복제 - 깊은 복제

- ▶ 얕은 복제는 참조 타입 필드의 경우, 주소만 복사되기 때문에 참조 타입의 필드는 같은 주소를 갖게 된다
- ▶ 깊은 복제 : 참조하고 있는 객체도 복사하는 것
- ▶ Cloneable 인터페이스를 구현한 후, `clone` 메서드를 오버라이드 하여 내부의 참조 객체도 복제해야 한다

```
public class Scoreboard implements Cloneable {  
    private int scores[];  
  
    // ...  
    @Override  
    protected Object clone() throws CloneNotSupportedException{  
        //먼저 얕은 복제를 시도  
        Scoreboard clone = (Scoreboard)super.clone();  
        //내부 참조 객체 복제 시도  
        clone.scores = Arrays.copyOf(scores, scores.length);  
        return clone;  
    }  
}
```

Java Basic APIs

String Class

Language API

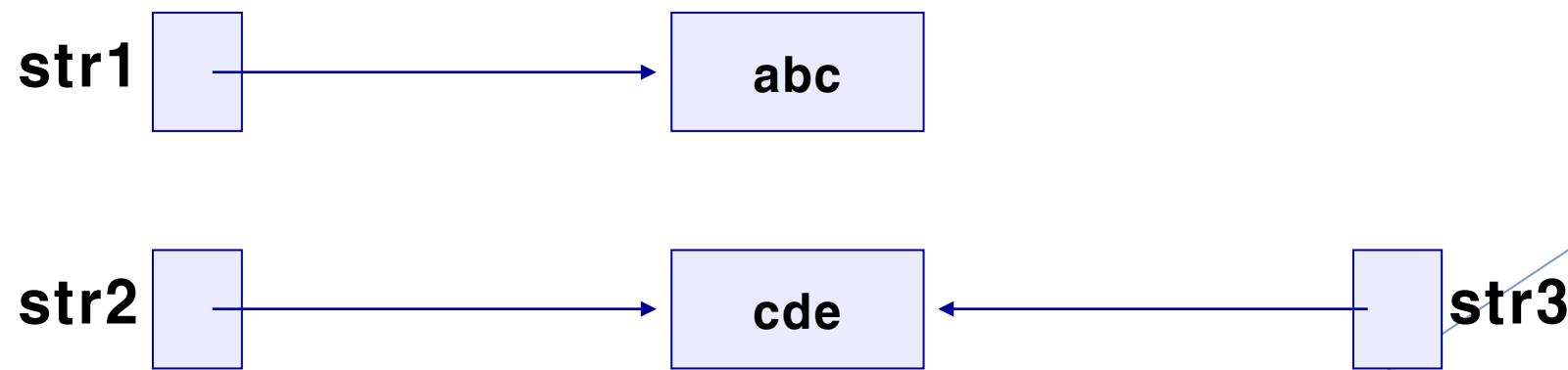
: String 클래스 - Character vs String

- ▶ 문자 (Character)
 - ▶ 기본 자료형 (char)
 - ▶ 문자 선언 `char letter;`
 - ▶ 문자 할당 `letter = 'A';`
 - ▶ 문자 사용 `System.out.println(letter); System.out.println((int)letter);`
- ▶ 문자열 (String) 클래스
 - ▶ 연속된 문자 (character)들을 저장하고 다루기 위한 클래스
 - ▶ 문자열 상수: “”로 둘러싸인 문자열 -> 한번 만들어진 String 객체는 변경 불가 (immutable)
 - ▶ 문자열 선언 `String greeting;`
 - ▶ 문자열 할당 `greeting = "Hello Java!";`
 - ▶ 문자열 사용 `System.out.println(greetin);`
 - ▶ 특수 문자의 표현 (Escape characters)
 - ▶ \', \", \\, \\n, \\r, \\t 등

Language API

: String 클래스 - 메모리 그림 (I)

```
String str1;  
String str2, str3; // String 클래스변수 str1, str2, str3 선언  
str1 = "abc"; // str1은 생성된 String 클래스의 객체(Object)를 가리킴  
str2 = "cde"; // str2은 생성된 String 클래스의 객체(Object)를 가리킴  
str3 = str2; // str3에 str2의 주소 할당
```



Language API

: String 클래스 - String 연산

마지막 문자는 `length - 1`

Index	0	1	2	3	4	5	6	7	8	9	10
char	H	e	I	I	o		J	A	V	A	!

- ▶ “+” 연산자 : 문자열의 연결. 문자열과 다른 타입의 연결은 문자열로 변환되어 연결

```
String greet = "Hello";
String name = "JAVA";
System.out.println( greet + name + "!" );
System.out.println( greet + " " + name + "!" );
```

- ▶ Index

- ▶ String 객체 내의 문자 인덱싱은 배열과 같이 0부터 시작됨
- ▶ `charAt(position)` : 해당 위치의 문자를 반환
- ▶ `Substring(start, end)` : start부터 end까지의 문자들을 새로운 문자열로 반환

```
String greeting = "Hello JAVA!";
greeting.charAt(0);
greeting.charAt(10);
greeting.substring(1, 3);
```

Example

: String Class

Q) 메모리 그림과 API 문서를 활용해서 화면에 출력되는 값을 예상해 보십시오.

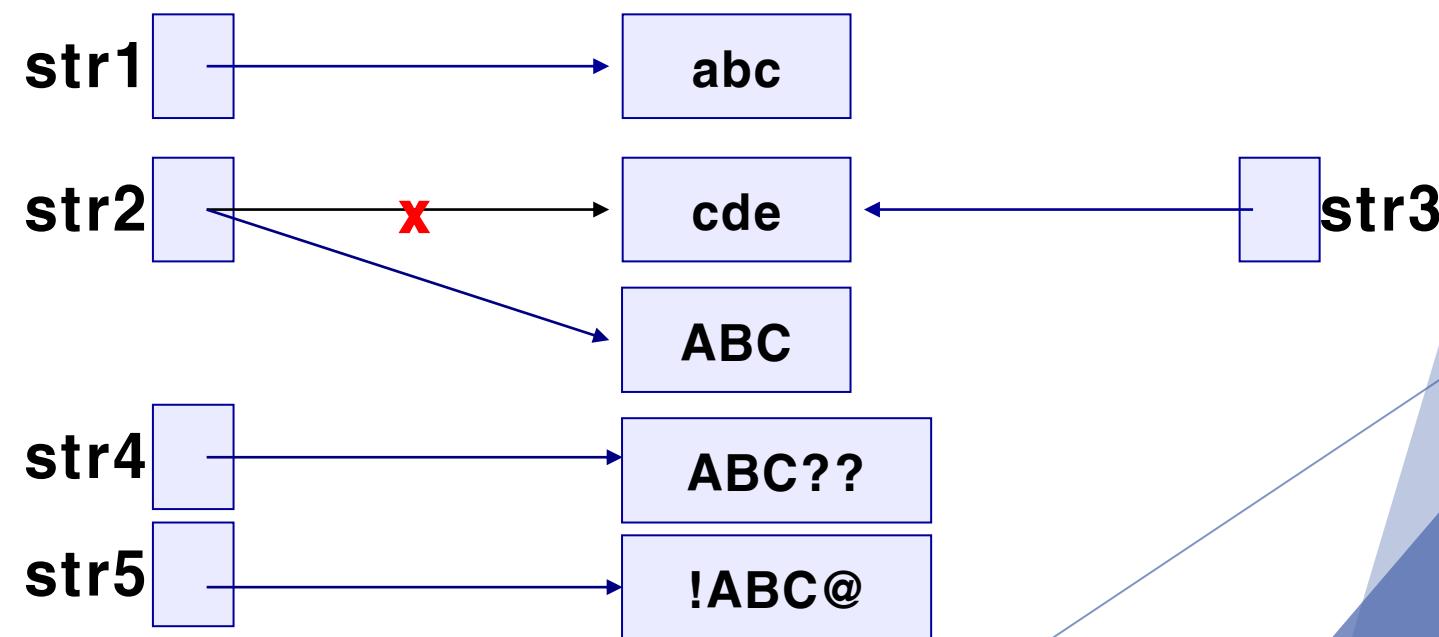
```
String str1 = "abc";
String str2;
str2 = str1.toUpperCase();
String str4 = str2.concat("??");
String str5 = "!.concat(str2).concat("@");

System.out.println("str1: " + str1);
System.out.println("str2: " + str2);
System.out.println("str4: " + str4);
System.out.println("str5: " + str5);
```

Language API

: String 클래스 - 메모리 그림 (II)

```
str2 = str1.toUpperCase();
String str4 = str2.concat( "??");
String str5 = "!".concat(str2).concat( "@" );
```

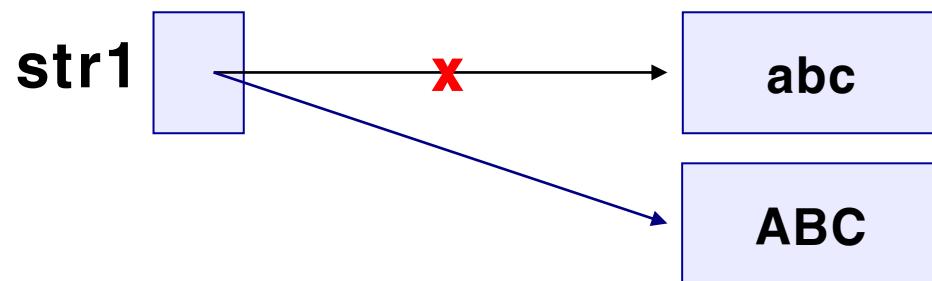


Language API

: String 클래스 - 메모리 그림 (III)

- ▶ Java의 String 객체는 한번 생성하면 변경할 수 없고 재할당시 새로운 String 클래스 객체가 생성

```
str1 = str1.toUpperCase();
```



Language API

: String 클래스 - 스트링 리터럴 vs new String()

```
public class StringTest {  
    public static void main( String[] args ) {  
        String s1 = "hello";  
        String s2 = "hello";  
  
        String s3 = new String("hello");  
        String s4 = new String("hello");  
  
        System.out.println( s1 == s2 );  
        System.out.println( s3 == s4 );  
        System.out.println( s2 == s4 );  
    }  
}
```

Language API

: String class methods

스트링 연관 메

public class String		
String(String s)		<i>create a string with the same value as s</i>
int length()		<i>number of characters</i>
char charAt(int i)		<i>the character at index i</i>
String substring(int i, int j)		<i>characters at indices i through (j-1)</i>
boolean contains(String substring)		<i>does this string contain substring?</i>
boolean startsWith(String pre)		<i>does this string start with pre?</i>
boolean endsWith(String post)		<i>does this string end with post?</i>
int indexOf(String pattern)		<i>index of first occurrence of pattern</i>
int indexOf(String pattern, int i)		<i>index of first occurrence of pattern after i</i>
String concat(String t)		<i>this string with t appended</i>
int compareTo(String t)		<i>string comparison</i>
String toLowerCase()		<i>this string, with lowercase letters</i>
String toUpperCase()		<i>this string, with uppercase letters</i>
String replaceAll(String a, String b)		<i>this string, with as replaced by bs</i>
String[] split(String delimiter)		<i>strings between occurrences of delimiter</i>
boolean equals(Object t)		<i>is this string's value the same as t's?</i>
int hashCode()		<i>an integer hash code</i>

실습예제

다음의 실행결과와 같이 출력하는 프로그램을 작성하세요.

- 1) “aBcAbCabcABC” 문자열 String 객체를 생성한다.
- 2) 3번째 문자열 출력한다
- 3) “abc”문자열이 처음으로 나타나는 인덱스를 추적한다.
- 4) 위 문자열의 문자 개수를 출력한다.
- 5) ‘a’를 ‘R’로 대체한 결과를 출력한다.
- 6) “abc”문자열을 “123”문자열로 대체한 결과를 출력한다.
- 7) 처음 3개의 문자열만을 출력한다.
- 8) 문자열을 모두 대문자로 변경하여 출력한다.

Language API

: StringBuffer 클래스

- ▶ 가변 크기의 버퍼를 가짐
- ▶ String이 immutable인 것에 비해, StringBuffer 객체는 수정 가능하다
 - ▶ 버퍼 크기는 문자열의 길이에 따라 가변적으로 변한다

```
public class SBTest {  
    public static void main( String[] args ) {  
        StringBuffer sb = new StringBuffer("This"); //생성  
        System.out.println( sb );  
        sb.append(" is pencil"); // 문자열 덧붙이기  
        System.out.println( sb );  
        sb.insert(7, " my"); // 문자열 삽입  
        System.out.println( sb );  
        sb.replace(7, 10, " your"); // 문자열 대치  
        System.out.println( sb );  
        sb.setLength(5); // 버퍼크기 조정  
        System.out.println( sb );  
    }  
}
```

Java Basic APIs

Arrays Class

Language API

: Arrays 클래스

- ▶ 배열을 조작하는 기능을 가진 API
 - ▶ 복사
 - ▶ 항목 정렬
 - ▶ 항목 검색

리턴 타입	메소드 이름	설명	http://palpit.tistory.com
int	binarySearch(배열, 찾는값)	전체 배열 항목에서 찾는 값이 있는 인덱스 리턴	
타겟 배열	copyOf(원본배열, 복사할 길이)	원본 배열의 인덱스 0에서 복사할 길이만큼 복사한 배열 리턴	
타겟 배열	copyOfRange(원본배열, 시작, 끝 인덱스)	원본 배열의 시작과 끝 인덱스까지 복사한 배열 리턴	
boolean	deepEquals(배열, 배열)	두 배열의 깊은 비교(중첩 배열 비교)	
boolean	equals(배열, 배열)	두 배열의 얕은 비교	
void	fill(배열, 값)	전체 배열 항목에 동일한 값을 저장	
void	fill(배열, 시작, 끝 인덱스, 값)	시작부터 끝 인덱스까지의 항목에만 값을 저장	
void	sort(배열)	배열의 전체 항목을 오름차순 정렬	
String	toString(배열)	“[값1, 값2, …]” 와 같은 문자열 리턴	

Java Basic APIs

Wrapper Class

Language API

: Wrapper 클래스

- ▶ 기본 데이터형을 객체로 다루기 위한 포장 클래스



byte	short	int	long	char	float	double	boolean
Byte	Short	Int	Long	Character	Float	Double	Boolean

- ▶ 사용 이유

- ▶ 자바에서는 객체를 대상으로 처리하는 경우가 많음
- ▶ 특정 클래스는 객체만을 다루기 때문에 기본 데이터 형을 사용할 수 없음
- ▶ 문자열을 기본 타입값으로 변환할 때도 유용
- ▶ 그외, 다른 타입으로부터의 변환 등 유용한 유틸리티 메서드 제공

Language API

: Wrapper 클래스 - 객체 생성

- ▶ new를 이용한 Wrapper 클래스 객체 생성은 Deprecated 되었음
- ▶ Java 9 이후부터는 내부의 static 메서드 valueOf를 이용하여 생성

```
public class WrapperClassTest {  
    public static void main( String[] args ) {  
        Integer i = Integer.valueOf( 10 );  
        Integer i2 = new Integer(20); //Deprecated  
        Character c = Character.valueOf( 'c' );  
        Float f = Float.valueOf( 3.14f );  
        Boolean b = Boolean.valueOf( true );  
        Integer i2 = Integer.valueOf( "10" );  
        Double d2 = Double.valueOf( "3.14" );  
        Boolean b2 = Boolean.valueOf( "false" );  
    }  
}
```

Language API

: Wrapper 클래스 - 활용

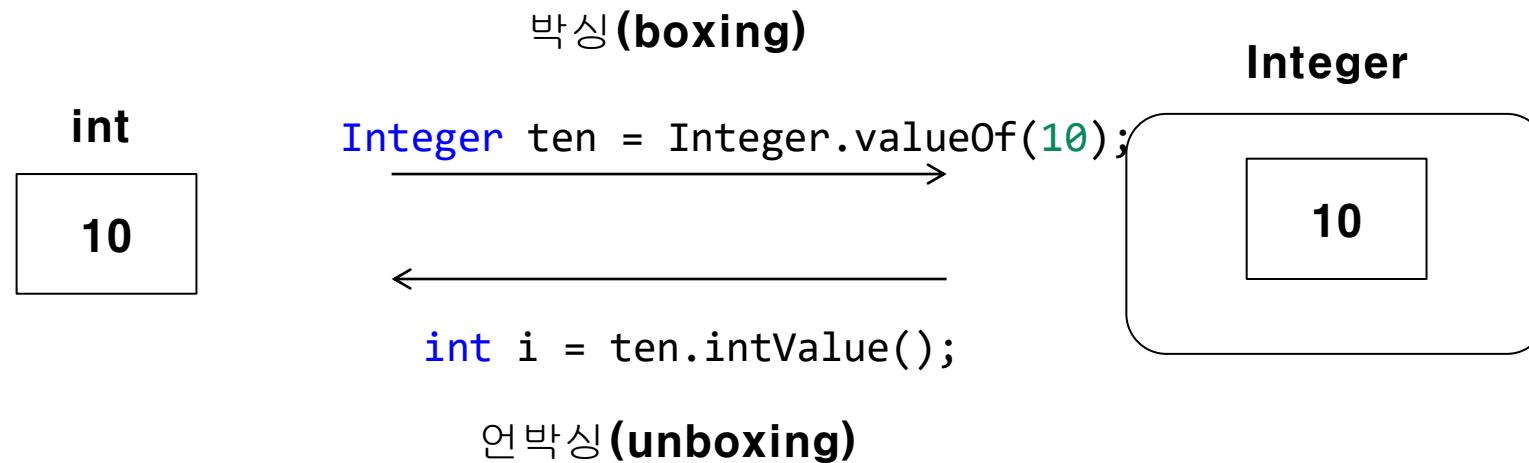
- ▶ 타입간 변환, 대부분 static 메서드
- ▶ 문자열을 기본 데이터 값으로 변환 또는 그 반대

```
public class WrapperTest {  
    public static void main( String[] args ) {  
        Integer i = Integer.valueOf( 10 );  
        Character c = Character.valueOf( '4' );  
        Double d = Double.valueOf( 3.1234566 );  
        System.out.println( Character.toLowerCase('A') ); // 대문자 A를 소문자로 변환  
        if( Character.isDigit( c ) ){ // 문자 c 가 숫자를 나타내면  
            System.out.println( Character.getNumericValue( c ) ); // 문자를 숫자로 변환  
        }  
        System.out.println( Integer.parseInt( "-123" ) ); // 문자열을 정수로 변환  
        System.out.println( Integer.parseInt( "10", 16 ) ); // 16진수 문자열을 정수로 변환  
        System.out.println( Integer.toBinaryString( 28 ) ); // 2진수로 표현된 문자열로 변환  
        System.out.println( Integer.bitCount( 28 ) ); // 2진수에서 1의 개수  
        System.out.println( Integer.toHexString( 28 ) ); // 16진수 문자열로 변환  
        System.out.println( i.doubleValue() ); // 정수를 double로 변환  
        System.out.println( d.toString() ); // Double을 문자열로 변환  
        System.out.println( Double.parseDouble("44.13e-16" ) ); // 문자열을 double로 변환  
    }  
}
```

Language API

: Wrapper 클래스 - 박싱(boxing)과 언박싱(unboxing)

- ▶ 박싱 (boxing) : 기본 데이터를 Wrapper 클래스에 담는 것
- ▶ 언박싱 (unboxing) : 박싱의 반대



Language API

: Wrapper 클래스 - 자동 박싱(auto boxing)과 자동 언박싱(auto unboxing)

- ▶ JDK 1.5부터는 박싱과 언박싱이 자동으로 수행됨

```
public class WapperClassTest {  
    public static void main( String[] args ) {  
        Integer i = 10;  
        System.out.println( i );  
        int n = i + 10;  
        System.out.println( n );  
    }  
}
```

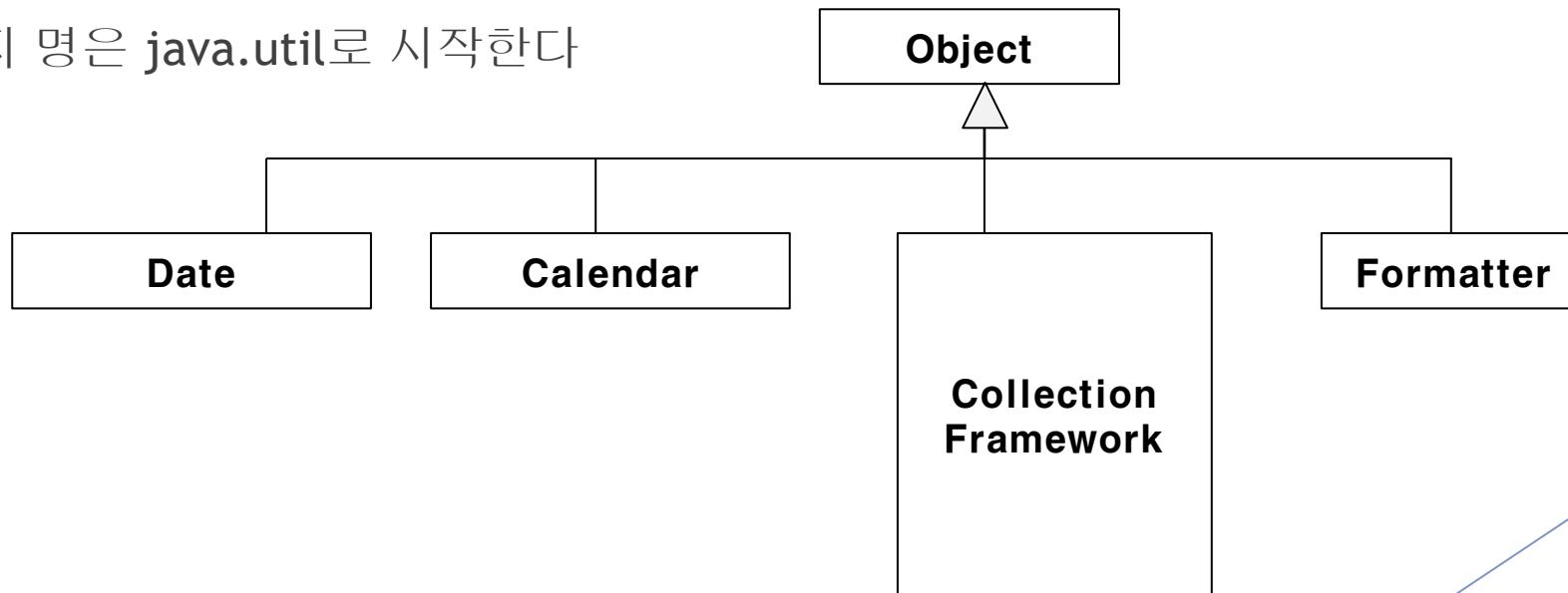
Java Basic APIs

Utility APIs

Java Utility API

: 개요

- ▶ 자바 프로그램에서 필요로 하는 편리한 기능을 모아둔 클래스들의 패키지
- ▶ 클래스 이름 앞에 패키지 이름을 사용하지 않았다면, `import`를 명시적으로 해야 하는 패키지
- ▶ 패키지 명은 `java.util`로 시작한다



Java Utility API

: Date 클래스

- ▶ 날짜와 시간에 관한 정보를 표현
- ▶ 많은 메서드들이 **deprecated** (폐지 예정)되고 **Calendar** 클래스에서 지원
 - ▶ 현재는 **Date()** 생성자만 주로 사용하고, 날짜에 관련된 기능들은 **Calendar** 클래스를 이용한다
- ▶ 만약 특정 포맷을 얻고 싶다면, **DateFormat** 클래스나 **SimpleDateFormat** 클래스를 임포트하여 사용한다

```
import java.text.DateFormat;
import java.util.Date;

public class DateTest {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Date now = new Date();
        System.out.println( now.toString() );
        System.out.println( now.toLocaleString() );      // Deprecated !!
    }
}

-----
DateFormat dateFormat1 = DateFormat.getDateInstance( DateFormat.FULL );
System.out.println( dateFormat1.format( now ) );

DateFormat dateFormat2 = DateFormat.getDateInstance( DateFormat.LONG );
System.out.println( dateFormat2.format( now ) );

DateFormat dateFormat3 =
DateFormat.getDateInstance( DateFormat.MEDIUM );
System.out.println( dateFormat3.format( now ) );

DateFormat dateFormat4 = DateFormat.getDateInstance( DateFormat.SHORT );
System.out.println( dateFormat4.format( now ) );
```

Java Utility API

: Date Format String

- ▶ `SimpleDateFormat` 클래스에 `Time Format`을 지정하여 원하는 형식의 날자 문자열을 얻을 수 있다

Format	
d	날짜의 한 자리 표현. 예) 5
dd	날짜의 두 자리 표현. 예) 05
M	월의 한 자리 표현. 예) 1
MM	월의 두 자리 표현. 예) 01
MMM	월의 세 자리 영어 표현. 예) Jan
MMMM	월의 전체 영어 표현. 예) January
yy	연도의 두 자리 표현. 예) 17
yyyy	연도의 네 자리 표현. 예) 2017

Format	
h	12시간 포맷의 한 자리 시간 표현. 예) 9
hh	12시간 포맷의 두 자리 시간 표현. 예) 09
H	24시간 포맷의 두 자리 시간 표현. 예) 09 (AM 9)
HH	24시간 포맷의 두 자리 시간 표현. 예) 21 (PM 9)
m	한 자리 분 표현. 예) 9
mm	두 자리 분 표현. 예) 09
s	한 자리 초 표현. 예) 3
ss	두 자리 초 표현. 예) 03
a	am/pm 표현

Java Utility API

: Calendar 클래스

- ▶ 날짜와 시간에 관한 정보를 표현
- ▶ `new`로 객체를 생성할 수 없고, `getInstance()` 메서드를 이용, `Calendar` 객체를 얻어 사용
- ▶ 날짜와 시간 표현을 위한 다양한 상수 제공

Java Utility API

: Calendar

▶ Calendar 클래스의 주요 상수

상수	사용방법	설명
static int YEAR	Calendar.YEAR	현재 년도를 가져온다.
static int MONTH	Calendar.MONTH	현재 월을 가져온다. (1월은 0)
static int DATE	Calendar.DATE	현재 월의 날짜를 가져온다.
static int WEEK_OF_YEAR	Calendar.WEEK_OF_YEAR	현재 년도의 몇째 주
static int WEEK_OF_MONTH	Calendar.WEEK_OF_MONTH	현재 월의 몇째 주
static int DAY_OF_YEAR	Calendar.DAY_OF_YEAR	현재 년도의 날짜
static int DAY_OF_MONTH	Calendar.DAY_OF_MONTH	현재 월의 날짜 (DATE와 동일)
static int DAY_OF_WEEK	Calendar.DAY_OF_WEEK	현재 요일 (일요일은 1, 토요일은 7)
static int HOUR	Calendar.HOUR	현재 시간 (12시간제)
static int HOUR_OF_DAY	Calendar.HOUR_OF_DAY	현재 시간 (24시간제)
static int MINUTE	Calendar.MINUTE	현재 분
static int SECOND	Calendar.SECOND	현재 초

Java Utility API

: Calendar

▶ Calendar 클래스 주요 메서드

Calendar 클래스의 메소드	설명
boolean after(Object when)	when과 비교하여 현재 날짜 이후이면 true, 아니면 false를 반환한다.
boolean before(Object when)	when과 비교하여 현재 날짜 이전이면 true, 아니면 false를 반환한다.
boolean equals(Object obj)	같은 날짜값인지 비교하여 true, false를 반환한다.
int get(int field)	현재 객체의 주어진 값의 필드에 해당하는 상수 값을 반환한다. 이 상수값은 Calendar 클래스의 상수중에 가진다.
static Calendar getInstance()	현재 날짜와 시간 정보를 가진 Calendar 객체를 생성한다.
Date getTime()	현재의 객체를 Date 객체로 변환한다.
long getTimelnMills()	객체의 시간을 1/1000초 단위로 변경하여 반환한다.
void set(int field, int value)	현재 객체의 특정 필드를 다른 값으로 설정한다.
void set(int year, int month, int date)	현재 객체의 년, 월, 일 값을 다른 값으로 설정한다.
void set(int year, int month, int date, int hour, int minute, int second)	현재 객체의 년, 월, 일, 시, 분, 초 값을 다른 값으로 설정한다.
void setTime(Date date)	date 객체의 날짜와 시간 정보를 현재 객체로 생성한다.
void setTimeInMills(long mills)	현재 객체를 1/1000초 단위의 주어진 매개변수 시간으로 설정한다.
int getActualMinimum(int field)	현재 객체의 특정 필드의 최소 값을 반환한다.
int getActualMaximum(int field)	현재 객체의 특정 필드의 최대 값을 반환한다.

Java Utility API

: Calendar 클래스 - 활용 I

```
import java.util.Calendar;
public class CalendarTest {
    public static void main(String[] args) {
        Calendar caledar = Calendar.getInstance(); // 지금!
        printDate(caledar);
        caledar.set(Calendar.YEAR, 2016); // 년도를 2016년으로 설정
        printDate(caledar); // 년,월,일을 설정함(2016-2-12), 월은 0부터 시작하므로 -1을 해줘야한다.
        caledar.set(2016, 1, 12);
        printDate(caledar);
        caledar.set(2016, 1, 12, 13, 59); // 년,월,일,시,분을 설정(2016-2-12 오후 1:59)
        printDate(caledar);
        caledar.set(2016, 5, 12, 13, 59, 30); // 년,월,일,시,분,초를 설정(2016-6-12 오후 1:59:30)
        printDate(caledar);
    }

    public static void printDate(Calendar caledar) {
        System.out.println(caledar.get(Calendar.YEAR) + "년 "
            + (caledar.get(Calendar.MONTH) + 1) + "월 " // 0부터 시작함
            + caledar.get(Calendar.DATE) + "일 "
            + (caledar.get(Calendar.AM_PM) == 0 ? "오전 " : "오후 ") // 오전:0, 오후:1
            + caledar.get(Calendar.HOUR) + "시 "
            + caledar.get(Calendar.MINUTE) + "분 "
            + caledar.get(Calendar.SECOND) + "초");
    }
}
```

Java Utility API

: Calendar 클래스 - 활용 II

```
import java.util.Calendar;

public class CalendarTest {

    public static void main(String[] args) {
        // 오늘
        Calendar caledar = Calendar.getInstance();
        printDate(caledar);
        // 100일 후
        caledar.add(Calendar.DATE, 100);
        printDate(caledar);
        // 다시 오늘
        caledar = Calendar.getInstance();
        printDate(caledar);
        // 10달전
        caledar.add(Calendar.MONTH, -10);
        printDate(caledar);
        // 다시 오늘
        caledar = Calendar.getInstance();
        printDate(caledar);
        // 오늘은 올해 몇 일째 되는날?
        System.out.println("오늘은 올해 " + caledar.get(Calendar.DAY_OF_YEAR) + "일째 되는 날입니다.");
    }
}
```

Java Utility API

: Formatter 클래스

- ▶ 데이터의 형식화된 문자열을 생성하는 클래스
- ▶ 형식 문자열 (format 지정)
- ▶ 문자열 생성을 위하여 Appendable 인터페이스를 구현한 클래스 객체 (StringBuffer 객체)를 Formatter 생성자에서 등록한다

```
StringBuffer sb = new StringBuffer();
Formatter f = new Formatter( sb );
String name = "Simon Readmore";
int score = 100;
f.format( "%s님의 점수는 %d 입니다.", name, score );
```

```
System.out.println(f);
```

```
String s = String.format("%s님의 점수는 %d 입니다.", name, score );
```

Java Basic APIs

Generic

제네릭(Generic)

- ▶ 클래스 내부에서 사용할 데이터 타입을 외부에서 지정하는 기법
(클래스 내부에서 사용할 데이터 타입을 나중에 인스턴스를 생성할 때 확정하여 사용)
- ▶ 제네릭을 사용해야 하는 이유
 - ▶ 객체의 타입을 컴파일시에 강하게 체크할 수 있음
 - ▶ 형변환의 번거로움이 줄어든다

개별 타입만 다룰 수 있는 클래스

StringBox.java, Int.java

```
public class StringBox {  
    String content;  
    public String getContent() {  
        return content;  
    }  
    public void setContent(String content)  
    {  
        this.content = content;  
    }  
}
```

모든 타입을 다룰 수 있는 클래스

ObjList.java

```
public class ObjList {  
    private Object[] pArray;  
    private int crtPos;  
    public ObjList() {  
        this.pArray = new Object[3];  
        this.crtPos = 0;  
    }  
    public void add(Object o) {  
        pArray[crtPos] = o;  
        crtPos++;  
    }  
}
```

사용 타입을 나중에 확정
제네릭 클래스

GenericList.java

```
public class GenericList<T> {  
    private T[] pArray;  
    private int crtPos;  
    public GenericList() {  
        this.pArray = new T[3];  
        this.crtPos = 0;  
    }  
    public void add(T o) {  
        pArray[crtPos] = o;  
        crtPos++;  
    }  
}
```

제네릭(Generic)

- ▶ 제네릭 타입 : 타입을 파라미터로 가지는 클래스와 인터페이스
 - ▶ 기초 자료형은 제네릭 타입으로 사용할 수 없다.

```
public class 클래스명<T> { ... }  
public interface 인터페이스명<T> { //... }
```

GenericBox.java

```
public class GenericBox<T> {  
    T content;    설계시에는 내부에서 사용할 타입을 명시하지 않음  
  
    public T getContent() {  
        return content;  
    }  
  
    public void setContent(T content) {  
        this.content = content;  
    }  
}  
타입 파라미터의 개수는 제한하지 않음
```



```
Box<Integer> intBox = new Box<Integer>();  
Box<String> strBox = new Box<String>();
```

제네릭(Generic)

- ▶ 제네릭의 상속

- ▶ 제네릭 클래스들도 클래스이므로 상속 구조를 나타낼 수 있다
- ▶ 제네릭 클래스들 간의 상속은 `extends`나 `implements`로 표시한다

- ▶ 와일드 카드

- ▶ 기호 ?로 나타내며 어떠한 타입도 될 수 있다.
- ▶ `<?>` : 모든 타입 가능. == `<? extends Object>`
- ▶ `<? extends T>` : T와 T를 상속받은 자손들만 가능
- ▶ `<? super T>` : T와 T의 조상들만 가능

Java Basic APIs

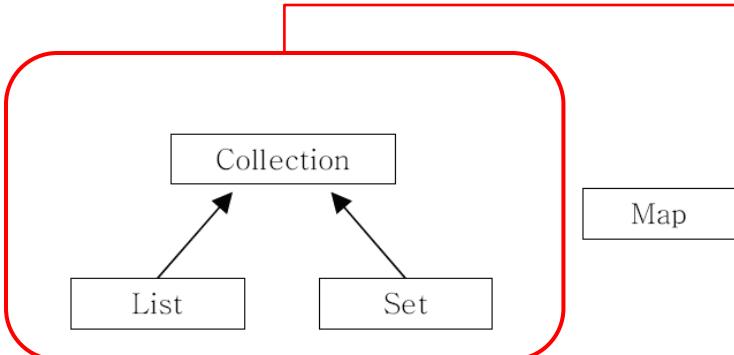
Collection Framework

Collection Framework

- ▶ 컬렉션 (Collection)
 - ▶ 다수의 데이터, 즉 데이터 그룹을 의미
- ▶ 프레임워크 (Framework)
 - ▶ 표준화, 정형화된 체계적인 프로그래밍 방식
- ▶ 컬렉션 프레임워크
 - ▶ 컬렉션을 저장하는 클래스들을 표준화한 설계
 - ▶ 다수의 데이터를 쉽게 처리할 수 있는 방법을 제공하는 클래스들로 구성
 - ▶ JDK 1.2부터
- ▶ 컬렉션 클래스 (Collection Class)
 - ▶ 다수의 데이터를 저장할 수 있는 클래스
 - ▶ Vector, ArrayList, HashSet 등

Collection Framework

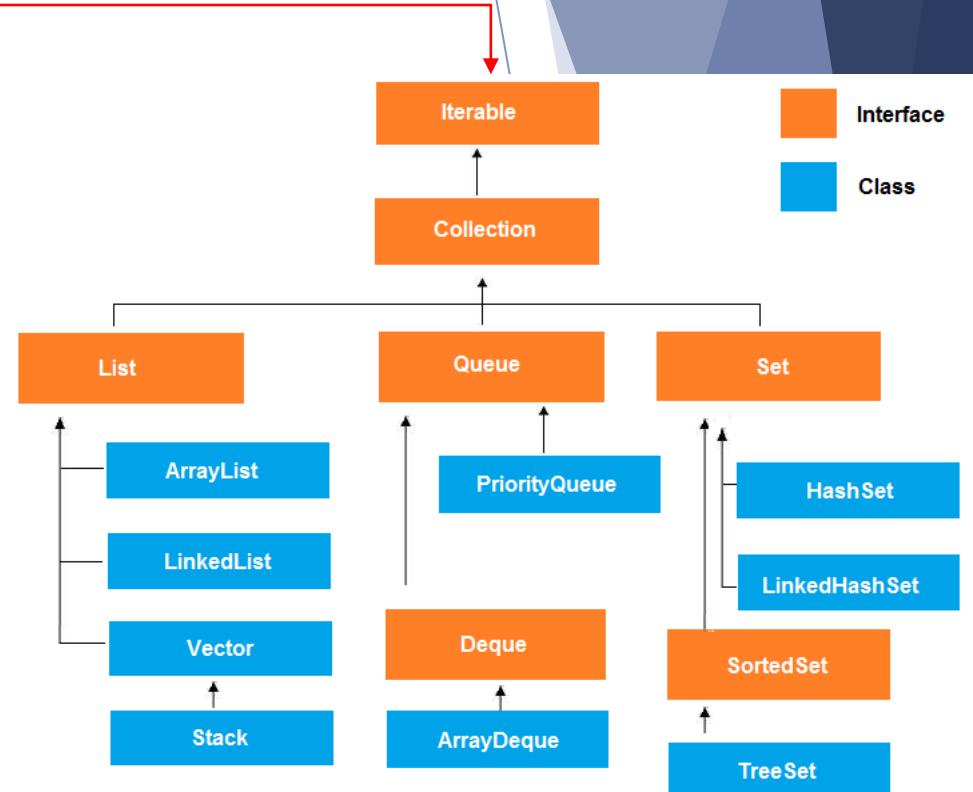
: 핵심 인터페이스



[그림11-1] 컬렉션 프레임워크의 핵심 인터페이스간의 상속계층도

인터페이스	특 징
List	순서가 있는 데이터의 집합. 데이터의 중복을 허용한다. 예) 대기자 명단 구현클래스 : ArrayList, LinkedList, Stack, Vector 등
Set	순서를 유지하지 않는 데이터의 집합. 데이터의 중복을 허용하지 않는다. 예) 양의 정수집합, 소수의 집합 구현클래스 : HashSet, TreeSet 등
Map	키(key)와 값(value)의 쌍(pair)으로 이루어진 데이터의 집합 순서는 유지되지 않으며, 키는 중복을 허용하지 않고, 값은 중복을 허용한다. 예) 우편번호, 지역번호(전화번호) 구현클래스 : HashMap, TreeMap, Hashtable, Properties 등

[표11-1] 컬렉션 프레임워크의 핵심 인터페이스와 특징



Collection Framework

: 배열 (Array) vs 리스트 (List)

	배열(Array)	리스트(List)
장점	<ul style="list-style-type: none">▪빠른 접근 (faster access)▪기본 자료유형 사용 가능	<ul style="list-style-type: none">▪가변적인 크기▪필요시 메모리를 할당하므로 효율적
단점	<ul style="list-style-type: none">▪고정된 크기▪비효율적 메모리 점유▪최대 크기를 넘어서는 사용을 위해서는 배열을 새로 정의해야 함	<ul style="list-style-type: none">▪느린 접근 (slower access)▪참조자료 유형만 사용 가능
사용	<ul style="list-style-type: none">▪Java에서 자체적으로 지원	<ul style="list-style-type: none">▪<code>java.util.List</code> 인터페이스 정의

Collection Framework

: Vector

- ▶ **Vector** 클래스 메서드 (Method)
 - ▶ 생성자 : `Vector()` / `Vector(int capacity)`
 - ▶ 개체 추가 : `addElement(Object o)` / `insertElementAt(Object o, int index)`
 - ▶ 개체 참조 : `elementAt(int index)`
 - ▶ 역개체참조 : `indexOf(Object o)`
 - ▶ 개체 변경 : `setElementAt(Object o, int index)`
 - ▶ 개체 삭제 : `remove(Object o)` / `remove(int index);`
 - ▶ 객체 검색 : `contains(Object o)`
 - ▶ 크기와 용량 : `size()` / `capacity()`
 - ▶ 비우기 : `clear()`
 - ▶ 복사 : `clone()`

Collection Framework

: Vector Example

```
public class VectorTest {  
    public static void main(String[] args) {  
        Vector v = new Vector();  
        System.out.println("Size=" + v.size() + " Capacity=" + v.capacity());  
  
        for( int i=0; i<10; i++ ) {  
            v.addElement(Integer.valueOf(i));  
        }  
        System.out.println("Size=" + v.size() + " Capacity=" + v.capacity());  
  
        Integer i0 = (Integer)v.elementAt(0);  
        int i1 = (Integer)v.elementAt(9);  
        System.out.println(i1);  
  
        v.removeElement(i0);  
        v.removeElement(1);  
        System.out.println(v);  
    }  
}
```

Collection Framework

: Vector Class

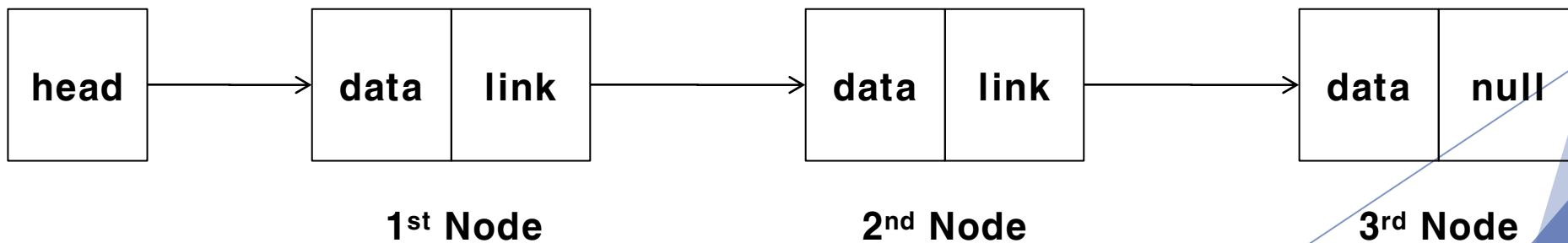
- ▶ 객체만 저장할 수 있음
- ▶ Vector의 참조 결과는 항상 Object 타입 -> 적절한 Type으로 변환 후 사용
- ▶ Generics로 지정하지 않으면 여러 타입을 동시에 저장 가능
 - ▶ 꺼낸 객체의 타입을 알고 있어야 함
 - ▶ 혹은 instanceof 연산자로 확인 후 사용

```
Vector v = new Vector();
v.addElement(1); //1은 int 기본자료형이지만 자바 버전에 따라 Packing되어 삽입된다
v.addElement(Integer.valueOf(1)); //int에 대응되는 Wrapper 클래스인 Integer 사용
v.addElement(Double.valueOf(3.14)); //어떤 타입의 객체도 저장 가능
Integer i = v.elementAt(0); //elementAt 메소드의 반환타입은 Object 이므로 컴파일 안됨
Integer i = (Integer)v.elementAt(0); //명시적 내림변환 필요(downcasting)
Double d = (Double)v.elementAt(1);
```

Collection Framework

: Linked List

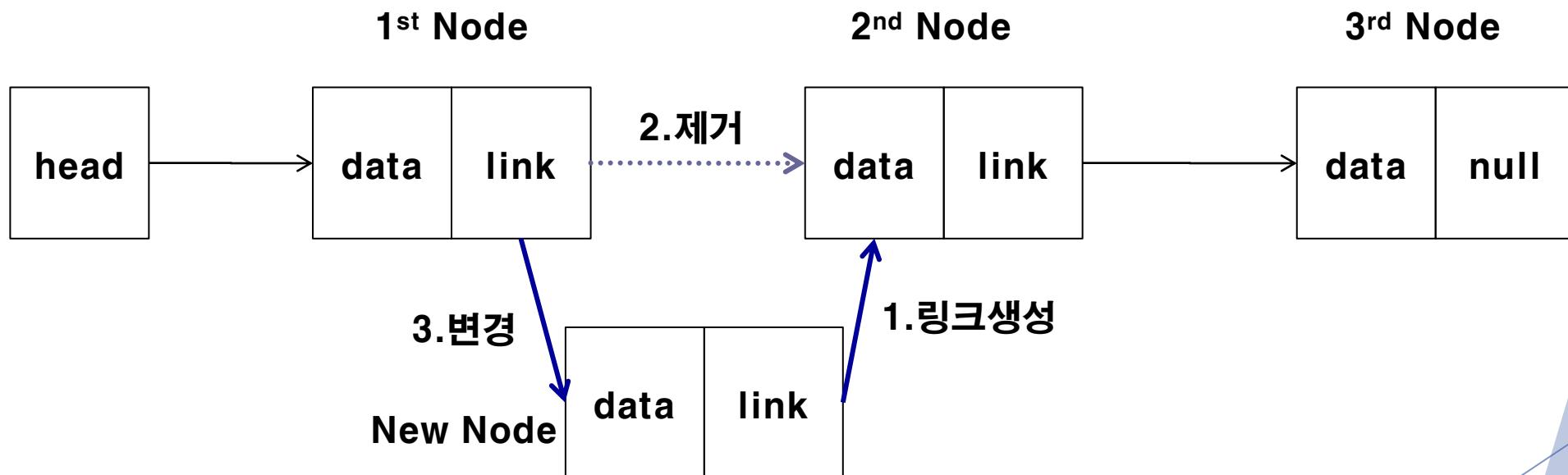
- ▶ 링크(Link)로 연결된 노드(Node)의 집합
- ▶ `java.util.LinkedList`
- ▶ 임의의 객체를 리스트로 저장하는 기능을 제공
- ▶ Index를 통한 참조 접근은 불가
 - ▶ Head로부터 링크를 따라가면서 접근
- ▶ 각 노드는 자신이 나타내는 데이터와 다음 노드(Node)로의 링크(Link)를 가지고 있음



Collection Framework

: Linked List

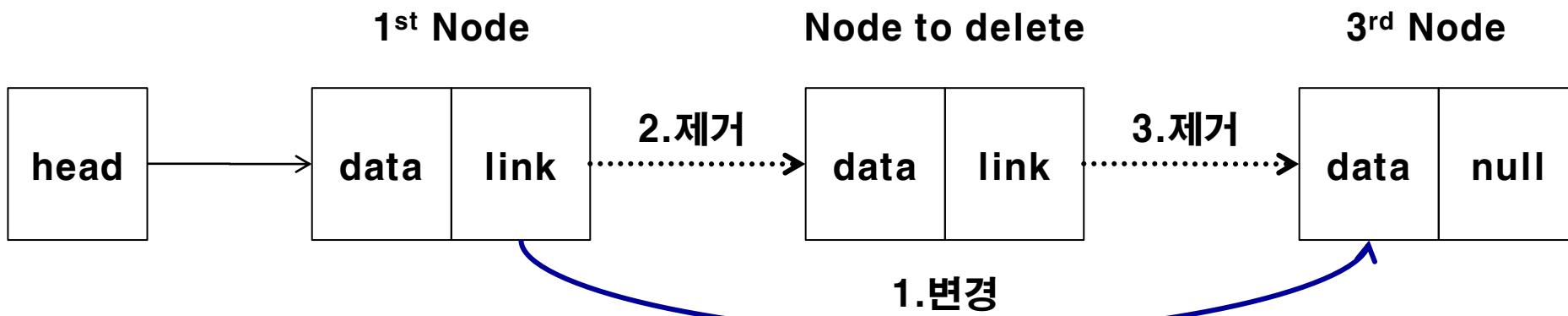
- ▶ 새로운 노드 추가



Collection Framework

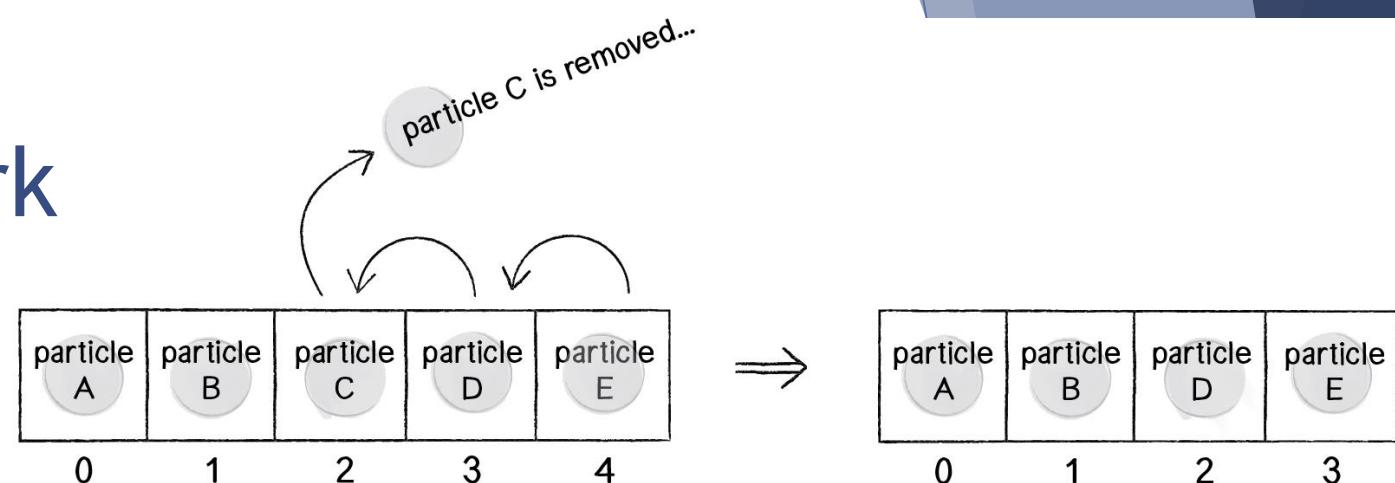
: Linked List

- ▶ 기존 노드 제거



Collection Framework

: Array List

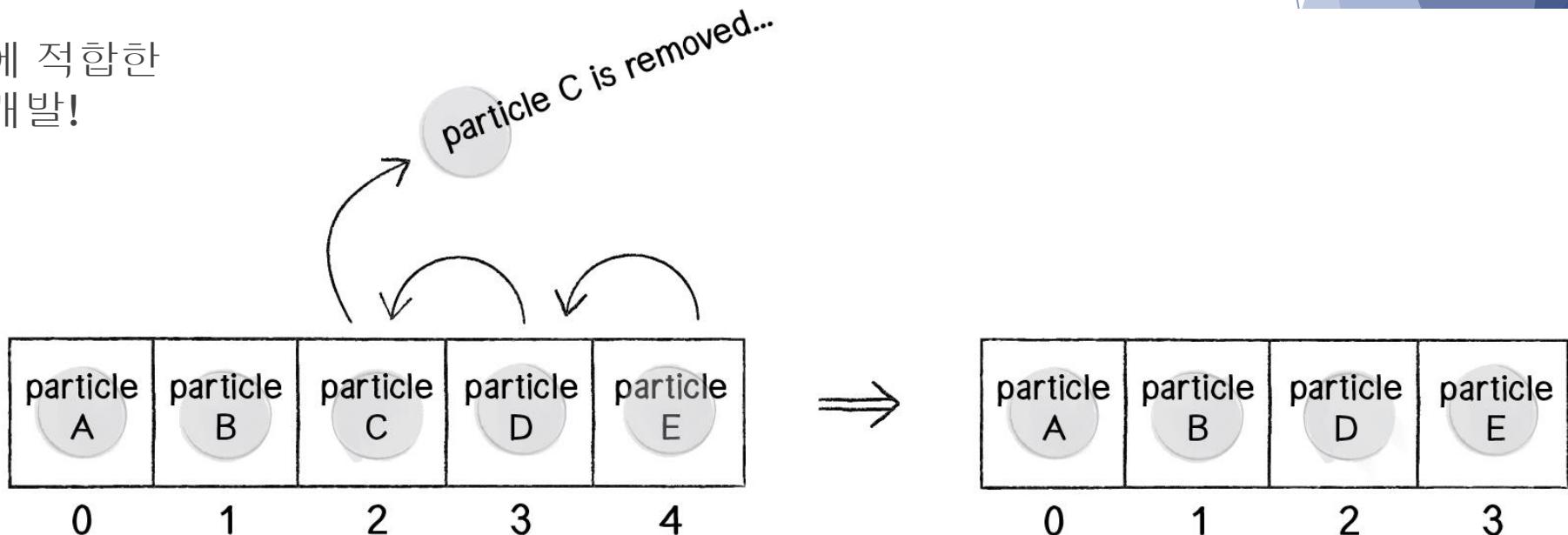


- ▶ java.util.ArrayList에 구현
- ▶ Vector, LinkedList, ArrayList 모두 추상 클래스 **AbstractList**를 상속받은 동적 자료 구조
- ▶ 배열을 다루는 것과 유사한 방식으로 동적 자료 구조를 사용할 수 있게 함
 - ▶ add(int index, E element), set(int index, E element), get(int index), remove(int index) methods
- ▶ **Vector vs List** : 항목 삽입, 삭제 동작이 동기화(synchronization) 여부의 차이
 - ▶ Vector : 동기화된 삽입 삭제 동작 제공
 - ▶ List : 삽입과 삭제가 동기화 되어 있지 않음 - 외부적 동기화 필요

Collection Framework

: Array List

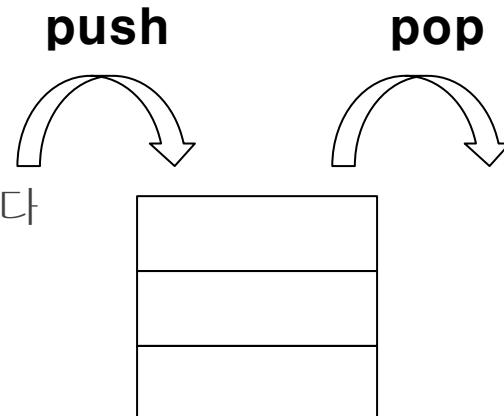
- ▶ `ArrayList`는 `LinkedList`와 동작 방식은 같으나 내부 구현 방식이 다르다
- ▶ `ArrayList`는 중간에 객체가 삭제되면 뒤의 객체들을 당겨, 인덱스를 재구성한다.
 - ▶ 따라서 마지막 인덱스에 객체가 추가되는 속도는 `LinkedList`보다 빠르지만 중간에 객체의 추가, 삭제가 빈번하게 일어나는 경우는 속도 저하가 일어나게 된다.
- ▶ 하고자 하는 작업에 적합한 자료구조를 택해 개발!



Collection Framework

: Stack

- ▶ `java.util.Stack` 클래스로 제공, `Vector` 클래스를 상속받아 구현
- ▶ 한쪽 끝점에서만 새로운 항목을 삽입, 기존 항목을 제거할 수 있음
 - ▶ Last In First Out (LIFO)
 - ▶ 쌓여있는 접시 : 새로운 접시를 위에 쌓거나 가장 위의 접시부터 사용 가능
- ▶ 스택에서의 메서드들
 - ▶ `push()` - 스택에 객체를 넣음
 - ▶ `pop()` - 스택에서 객체를 추출(`top`은 삭제)
 - ▶ `peek()` - `pop`과 같지만, `top` 값을 삭제하지 않는다
 - ▶ `empty()` - 스택이 비었는지 확인



Collection Framework

: Stack Example

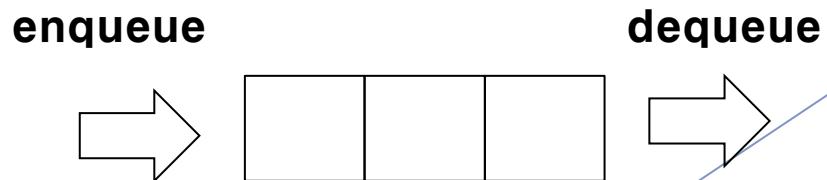
```
import java.util.Stack;
public class StackTest {
    public static void main(String[] args) {
        Stack s = new Stack();
        s.push("A");
        s.push("B");
        s.push("C");

        System.out.println(s.empty());
        System.out.println(s.pop());
        System.out.println(s.peek());
        System.out.println(s.pop());
    }
}
```

Collection Framework

: Queue

- ▶ java.util.Queue 클래스로 제공 (interface)
- ▶ 목록의 가장 마지막에 새로운 항목이 추가
- ▶ 기존 항목의 제거는 리스트의 처음에서 일어남
- ▶ First In First Out (FIFO)
- ▶ 큐에서의 메서드들
 - ▶ offer()
 - ▶ poll()
 - ▶ peek()



Collection Framework

: Queue Example

```
import java.util.LinkedList;
import java.util.Queue;
public class QueueTest {
    public static void main(String[] args) {
        Queue q = new LinkedList();

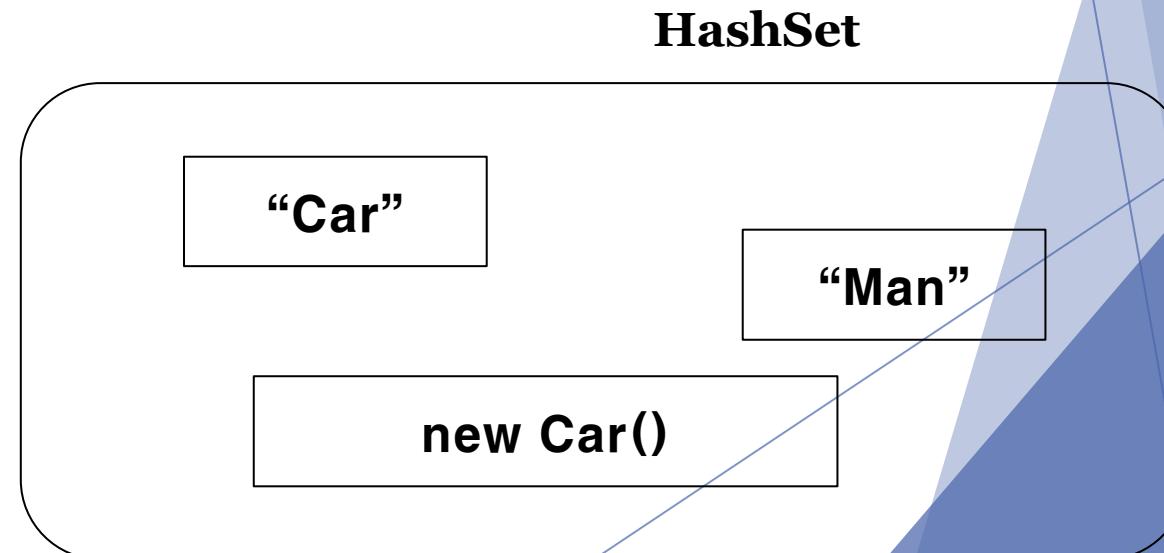
        q.offer("A");
        q.offer("B");
        q.offer("C");

        System.out.println(q.poll());
        System.out.println(q.peek());
        System.out.println(q.poll());
    }
}
```

Collection Framework

: Hash Set

- ▶ `java.util.HashSet` 클래스로 제공
- ▶ 자료 구조에 포함된 자료의 순서나 키에 상관없이 자료 전체를 하나의 셋으로 관리할 수 있도록 해주는 자료 구조
- ▶ 해시테이블에서 키 없이 값들만 존재하는 경우
- ▶ 자료의 해시로 유지, 검색이 빠르다
- ▶ `HashSet`에서 사용 가능한 메서드들
 - ▶ `add`
 - ▶ `remove`
 - ▶ `contains`



Collection Framework

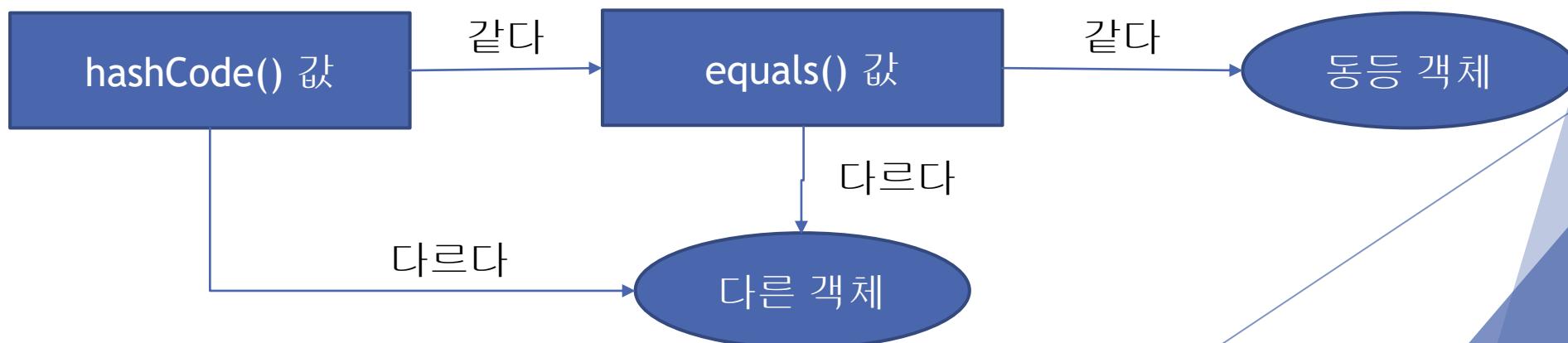
: Hash Set Example

```
public class HashSetTest {  
    public static void main(String[] args) {  
        HashSet hs = new HashSet();  
        hs.add("Car"); hs.add("Bus"); hs.add("Truck");  
        hs.add("Man"); hs.add("Woman"); hs.add("Child");  
        System.out.println(hs.size());  
        if( hs.contains("Man") ) {  
            hs.remove("Man");  
        }  
        if( hs.contains("Car") ) {  
            hs.add("Car2");  
        }  
        System.out.println(hs);  
    }  
}
```

Collection Framework

: hash 알고리즘과 hashCode()

- ▶ 객체 해시 코드란 객체를 식별할 하나의 정수값을 의미
- ▶ Object의 hashCode() 메서드는 객체의 메모리 번지를 이용하여 해시 코드를 만들어 리턴 -> 모든 객체는 다른 값을 가지고 있음
- ▶ Hash 관련 컬렉션들은 다음과 같은 방식으로 두 객체가 동등한지 아닌지를 비교 한다
 - ▶ equals와 hashCode 메서드를 이용, hash 알고리즘을 구현해 주어야 한다



Collection Framework

: Hash Table

- ▶ java.util.Hashtable 클래스
- ▶ 맵(Map) 인터페이스를 다루는 자료 구조
- ▶ 자료들의 순서가 아닌 키(key)와 값(value)의 쌍을 저장
- ▶ 키와 값은 모든 임의의 객체가 허용됨
- ▶ Hashtable에서 사용 가능한 메서드들
 - ▶ put
 - ▶ get
 - ▶ isEmpty
 - ▶ containsKey

key	value
“Car”	new Car()
“Man”	new Man()
“Cat”	new Cat()

Collection Framework

: Hash Table Example

```
public class HashTableTest {  
    public static void main(String[] args) {  
        Hashtable ht = new Hashtable();  
  
        Vector v1 = new Vector();  
        Vector v2 = new Vector();  
        v1.add("Taxi"); v1.add("Bus"); v1.add("Truck");  
        v2.add("Man"); v2.add("Woman"); v2.add("Child");  
        ht.put("Car",v1);  
        ht.put("Person",v2);  
        System.out.println(ht.get("Car"));  
  
        if( ht.containsKey("Person") ) {  
            System.out.println(ht.get("Person"));  
        }  
    }  
}
```

Collection Framework

: Enumeration and Iterator

- ▶ Enumeration : 벡터와 해시테이블에 존재하는 요소들에 대한 접근방식을 제공해주는 인터페이스
- ▶ Iterator: Collection Framework로 확장하면서 도입 (List, Set 등)
- ▶ 자바에서 제공하는 컬렉션에 대해 각 컬렉션의 항목들을 **순차적으로 접근**하는데 사용
- ▶ Enumeration
 - ▶ hasMoreElements()와 nextElement() 두 개의 메서드 제공
 - ▶ Vector::elements()
 - ▶ Hashtable::keys()
 - ▶ Hashtable::values()
- ▶ Iterator
 - ▶ hasNext(), next(), **remove()** 제공
 - ▶ Set::iterator()
 - ▶ List::iterator()

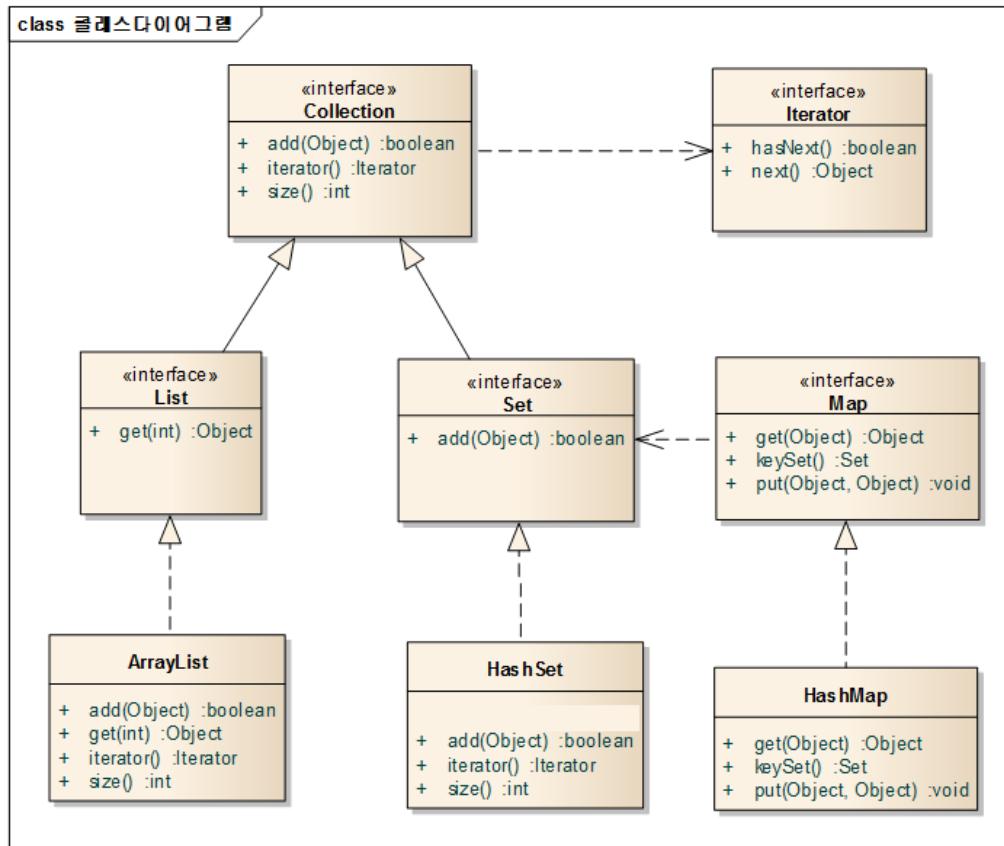
Collection Framework

: Enumeration and Iterator - Example

```
Vector v = new Vector();
// ...
Enumeration e = v.elements();
while(e.hasMoreElements()) {
    System.out.println((String)e.nextElement());
}
HashSet set = new HashSet();
// ...
Iterator iter = set.iterator();
while (iter.hasNext()) {
    i++;
    System.out.println(i + ":" + iter.next());
}
```

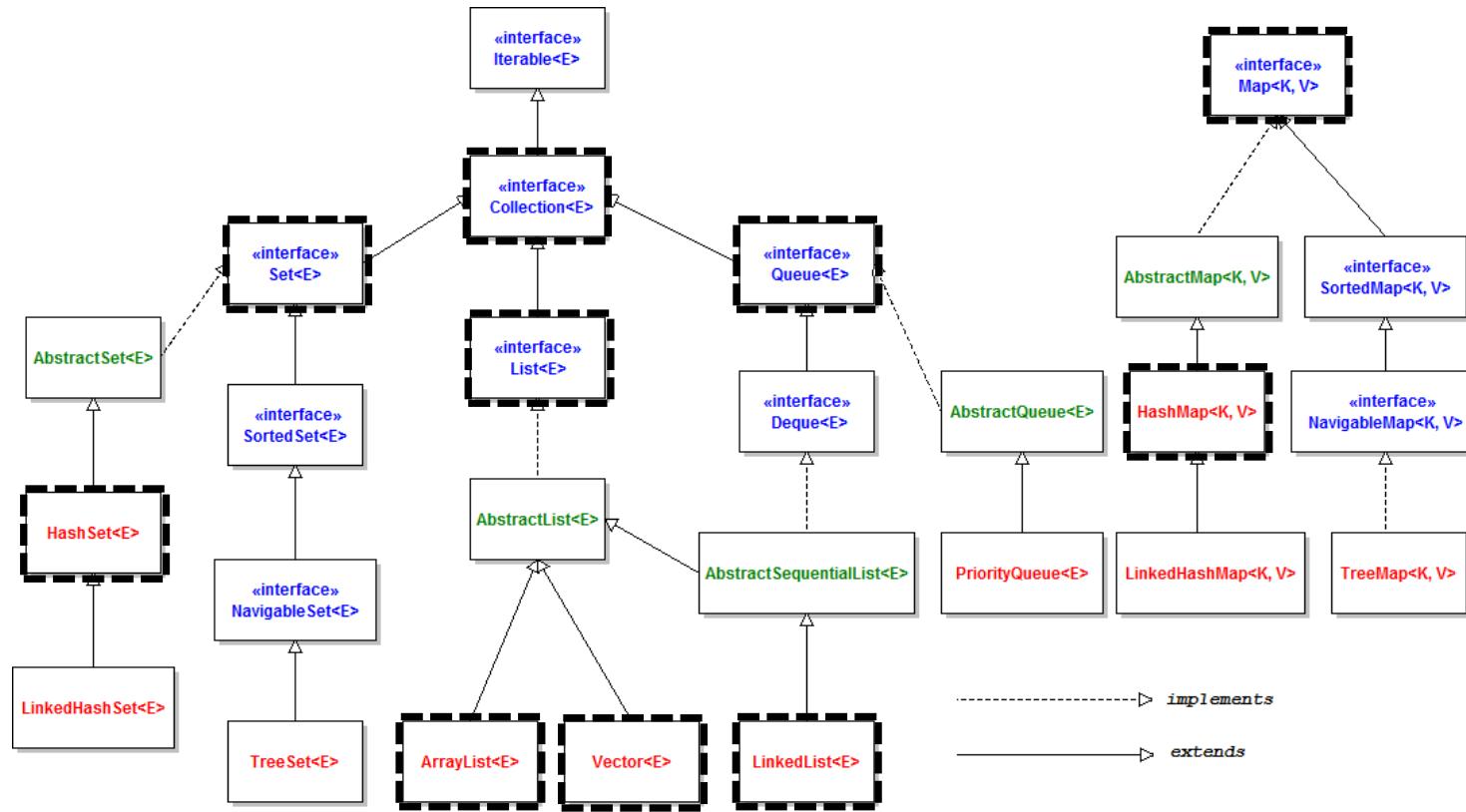
Collection Framework

: Collection Class Diagram I



Collection Framework

: Collection Class Diagram II

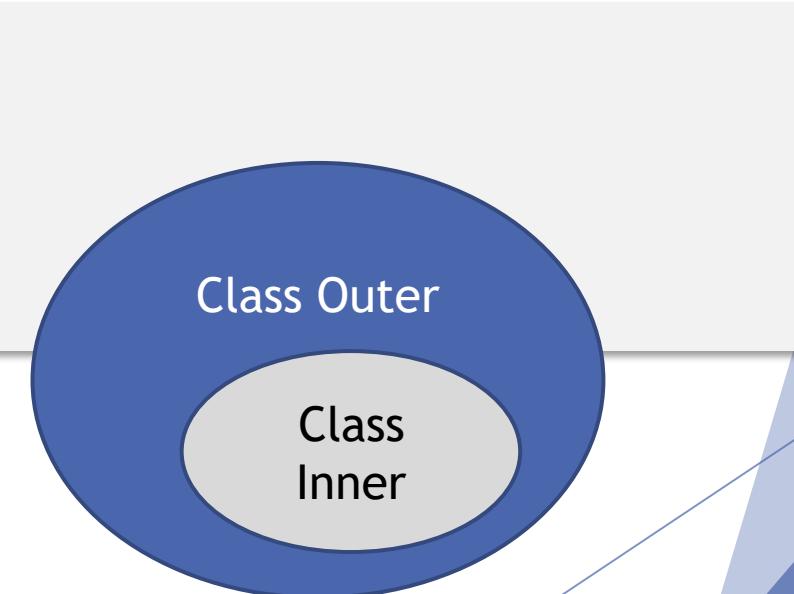


OOP Advanced

중첩 클래스

- ▶ 클래스가 여러 클래스와 관계를 맺지 않고, 특정 클래스와만 관계를 맺을 때는 관계 클래스를 클래스 내부에 선언하는 것이 좋다(Nested Class)
 - ▶ 중첩 클래스 사용의 이점
 - ▶ 두 클래스의 멤버들을 서로 쉽게 접근할 수 있다
 - ▶ 외부에는 불필요한 관계 클래스를 감춤으로써 코드의 복잡도를 줄여준다

```
class Outer {  
    // 외부 클래스  
    class Inner {  
        // 내부 클래스  
    }  
}
```



중첩 클래스

▶ 중첩 클래스 종류와 특징

분류		선언 위치	설명
멤버 클래스	인스턴스 멤버 클래스	<code>class A { class B { } }</code>	<ul style="list-style-type: none">- 외부 클래스의 멤버 변수 선언 위치에 선언- 내부에 정적 필드나 메서드는 선언할 수 없다- <code>class B</code>는 <code>A</code> 객체를 생성해야만 사용할 수 있다
	정적 멤버 클래스	<code>class A { static class B { } }</code>	<ul style="list-style-type: none">- 외부 클래스의 멤버 변수 선언 위치에 선언한다.- 내부에 모든 종류의 필드와 메서드를 선언할 수 있다.- <code>class B</code>는 <code>A</code> 클래스로 바로 접근할 수 있다.
로컬 클래스		<code>class A { void method() { class B { } } }</code>	<ul style="list-style-type: none">- 외부 클래스의 메서드 또는 초기화 블록 안에 선언한다.- 로컬 클래스는 접근제한자 및 <code>static</code>을 붙일 수 없다<ul style="list-style-type: none">- 메서드 내부에서만 사용되므로 접근을 제한할 필요가 없기 때문- <code>class B</code>는 <code>method()</code> 가 실행될 때만 사용할 수 있다.

▶ 중첩 클래스의 선언:

- ▶ 선언 위치에 따라 동일 위치의 변수와 동일한 유효범위와 접근성을 갖는다.

중첩 클래스

▶ 중첩 클래스 종류와 특징

분류		선언 위치	설명
멤버 클래스	인스턴스 멤버 클래스	<code>class A { class B { } }</code>	<ul style="list-style-type: none">- 외부 클래스의 멤버 변수 선언 위치에 선언- 내부에 정적 필드나 메서드는 선언할 수 없다- <code>class B</code>는 <code>A</code> 객체를 생성해야만 사용할 수 있다
	정적 멤버 클래스	<code>class A { static class B { } }</code>	<ul style="list-style-type: none">- 외부 클래스의 멤버 변수 선언 위치에 선언한다.- 내부에 모든 종류의 필드와 메서드를 선언할 수 있다.- <code>class B</code>는 <code>A</code> 클래스로 바로 접근할 수 있다.
로컬 클래스		<code>class A { void method() { class B { } } }</code>	<ul style="list-style-type: none">- 외부 클래스의 메서드 또는 초기화 블록 안에 선언한다.- 로컬 클래스는 접근제한자 및 <code>static</code>을 붙일 수 없다<ul style="list-style-type: none">- 메서드 내부에서만 사용되므로 접근을 제한할 필요가 없기 때문- <code>class B</code>는 <code>method()</code> 가 실행될 때만 사용할 수 있다.

▶ 중첩 클래스의 선언:

- ▶ 선언 위치에 따라 동일 위치의 변수와 동일한 유효범위와 접근성을 갖는다.

중첩 클래스

- ▶ 중첩 클래스에서 바깥 클래스 참조 얻기
 - ▶ 중첩 클래스에서 **this** 키워드는 중첩 클래스의 객체 참조
 - ▶ 중첩 클래스 내부에서 바깥쪽 클래스의 객체 참조를 얻으려면
 - ▶ 바깥클래스.this.필드명
 - ▶ 바깥클래스.this.메서드명()

```
class Outer {  
    // 외부 클래스  
    String field = "Outer Field";  
    class Nested {  
        // 중첩 클래스  
        String field = "Nested Field";  
        System.out.println(this.field);  
        System.out.println(Outer.this.field);  
    }  
}
```

익명 클래스

▶ 이름 없는(Anonymous) 클래스

- ▶ 클래스 선언과 동시에 단 한번만 사용 가능. 생성자를 선언할 수 없다.
- ▶ 단독으로 선언될 수 없고, 한 개의 클래스를 상속 받거나,
하나의 인터페이스만을 구현한다.

```
슈퍼클래스 변수 = new 슈퍼클래스() { ... } // 상속에 의한 익명 클래스  
인터페이스 변수 = new 인터페이스() { ... } // 구현에 의한 익명 클래스
```

- ▶ 주로 필드의 초기값이나 로컬 변수의 초기값, 매개변수의 매개값으로 주로 대입된다.
- ▶ UI 이벤트 처리나 쓰레드 객체를 간편하게 생성할 목적으로 많이 활용된다.

람다식

- ▶ 함수형 프로그래밍(Functional Programming)과 람다식
 - ▶ 함수형 프로그래밍:
 - ▶ 자료 처리를 수학적 함수의 계산으로 취급하고 상태와 가변 데이터를 멀리하는 프로그래밍 패러다임
 - ▶ 명령형 프로그래밍이 상태를 바꾸는 것을 강조하는 것과 달리, 함수형 프로그래밍에서는 **함수의 응용**을 강조한다.
-> 최근 함수형 프로그래밍의 여러 장점이 대두되면서 Java 8부터 함수형 프로그래밍을 지원
 - ▶ 람다식(Lambda Expression)
 - ▶ 자바에서 제공하는 함수형 프로그래밍 방식
 - ▶ 함수 이름이 없는 익명 함수를 만드는 것
 - ▶ 람다식은 메서드의 매개 변수로 전달될 수 있고 메서드의 결과로 반환될 수 있다
-> 메서드를 변수처럼 다루는 것이 가능해진다.

```
int add(int x, int y) {  
    return x + y;  
}
```



```
(int x, int y) -> { return x + y; }
```

람다식

▶ 람다식 문법 정리

▶ 매개변수 자료형과 괄호 생략

- ▶ 매개변수 자료형은 생략될 수 있고 매개변수가 하나인 경우는 괄호도 생략할 수 있다.

```
message -> { System.out.println(message); }
(x, y) -> { return x + y }
```

▶ 중괄호의 생략

- ▶ 함수 구현부가 한 문장인 경우 중괄호를 생략할 수 있다
- ▶ 구현부가 한 문장이더라도 return 문은 중괄호를 생략할 수 없다.

```
message -> System.out.println(message);
message -> return message.length(); // -> Error
```

▶ return문 생략

- ▶ 함수 구현부가 return문 하나라면 중괄호와 return문을 모두 생략할 수 있다.

```
message -> { return message.length(); } // -> OK
message -> message.length(); // -> OK
```

람다식

: 함수형 인터페이스

- ▶ 람다식은 메서드와 동등한 것이 아니라 익명 클래스의 객체와 동등하다.

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

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

- ▶ 람다식의 구현

- ▶ 람다식을 구현하기 위해서는 먼저 인터페이스를 선언하고 람다식으로 구현할 메서드를 선언
-> 함수형 인터페이스(@FunctionalInterface)

```
@FunctionalInterface  
public interface MyFunction {  
    int getResult(int num1, int num2); // 함수형 인터페이스는 오직 한 개의 메서드만 가진다  
}
```

람다식

: 함수형 인터페이스

▶ 람다식의 사용

- ▶ 람다식으로 정의된 익명 객체의 메서드를 호출하려면 참조 변수가 필요
- ▶ 이 때, 참조 변수의 타입은 람다식과 동등한 메서드가 정의되어 있는 클래스 혹은 인터페이스여야 한다.

```
// 익명 객체 방식
MyFunction add = new MyFunction() {
    @Override
    public int getResult(int x, int y) {
        return x + y;
    }
};
System.out.println(add.getResult(10, 20));
```

```
// Lambda
MyFunction max = (x, y) -> x >= y ? x : y;
System.out.println(max.getResult(10, 20));
```