[TODO] 주요 패키지와 클래스

- java.lang
  - java.lang 패키지는 자바에서 가장 기본적인 동작을 수행하는 클래스(Object 포함)들의 집합
    - 문자열, 수학 관련, 입출력 관련 클래스, 인터페이스와 같이 자바 프로그래밍에 과정에 필요한 기본적인 클래스와 인터페이스가 포함됨
  - 자동으로 모든 클래스와 인터페이스가 import 되므로 따로 import 문을 작성할 필요 없음
- java.util
  - 자바 유틸리티 패키지로 날짜, 시간 관련 클래스들과 벡터, 해시맵 등과 같은 다양한 콜렉션 클래스와 인터페이스를 제공함
- java.io, java.nio
  - 키보드, 모니터, 프린터, 디스크 등에 입출력을 할 수 있도록 도와줄 입출력 작업 클래스와 인터페이스 제공
  - nio 패키지는 "new io"의 줄임말로 성능을 개선한 입출력 작업을 진행할 수 있도록 도와줌
- java.awt, javax.swing
  - 자바 GUI 프로그래밍을 위한 클래스와 인터페이스 제공
- java.net
  - 네트워크 관련 작업(ex: 소켓 통신)을 위한 패키지
- java.util.concurrent
  - 동시성과 스레드 관련 작업 관련 패키지
- java.security
  - 보안과 관련된 해시 함수 및 대칭, 비대칭 키 생성과 관련된 패키지

Object 클래스

- java.lang 패키지에 포함되며 java.lang 패키지에 속한 클래스 중 가장 많이 사용되는 클래스
- Object 클래스는 **모든 자바 클래스의 최고 조상 클래스**
    - 따라서 자바의 모든 클래스는 Object 클래스의 모든 메소드를 바로 사용 가능함
    - Object 클래스에는 모든 객체가 공통으로 가져야 할 필수 메서드가 포함됨

Object 클래스의 주요 메서드

equals

- 객체가 **"논리적"으로 동등한지 비교**(=객체 내용을 비교)하기 위해서 사용하는 메서드
    - 주의) 만약 equals 메서드를 오버라이드했다면 **반드시 hashCode 메서드도 같이 오버라이드** 해야 함
- 두 객체가 **같은 리퍼런스를 가진 객체인지 비교**하기 위해서는 동등 비교 연산자(==)를 사용해야 함

equals 메서드의 기본 구현

```
public boolean equals(Object obj) {
    // 기본 동작 : 객체의 리퍼런스를 비교
    return (this == obj);
}
```

- 일반적인 오버라이드된 equals 메서드의 구현 방법은 다음과 같음

equals 메서드 오버라이딩할 클래스 정의

```
class Student {
    private String id;
    private String name;
    private int age;
    private String gender;
    private String hobby;

    public Student(String id, String name, int age, String gender, String hobby)
{
        this.id = id;
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.hobby = hobby;
    }

    // ...
}
```

자동 생성한 equals 메서드 구현 살펴보기

```
@Override
public boolean equals(Object o) {
    // 객체의 리퍼런스가 같다면 같은 객체
    if (this == o) return true;
    // 비교할 객체가 null이거나 클래스가 다르면 다른 객체
    if (o == null || getClass() != o.getClass()) return false;
    // 타입 변환 후
    Student student = (Student) o;
    // 모든 필드가 동일한지 값 및 (객체의 경우) 논리 비교
    return age == student.age && Objects.equals(id, student.id) &&
Objects.equals(name, student.name) && Objects.equals(gender, student.gender) &&
Objects.equals(hobby, student.hobby);
}
```

- 보통 **모든 필드 값이 같아야 같다고 판단하도록 구현하는 것이 일반적**이지만, 100%는 아니므로 개발자가 논리적으로 동등한지 여부를 따져봐야 함
    - 만약 Student 클래스의 equals 메서드를 호출하는데 다음과 같은 경우라면?

```
Student s1 = new Student("A1234", "김미림", 19, "여성", "게임");
Student s2 = new Student("A1234", "김미림", 19, "여성", "낚시");
// 결과는 false이지만, 이것이 올바른 판단인가?
System.out.println(s1.equals(s2));
```

- 만약 위의 경우에서 학번(id)만 같으면 논리적으로 동등한 Student라고 판단했다면 틀린 결정인가?
    - 특히 데이터베이스와 관련된 엔티티 클래스 작성에서 이러한 점을 주의해야 함

hashCode

- 해당 객체를 **가급적 유일하게 구분**지을 수 있도록 도와줄 정수값
    - 가급적 유일하다는 말은 유일하지 않은 값(=즉, 서로 겹치는 값)이 나올 수도 있다는 의미임

자동 생성한 hashCode 메서드 구현 살펴보기

```
@Override
public int hashCode() {
    return Objects.hash(id, name, age, gender, hobby);
}
```

메서드 구현 살펴보기

```
// Objects 클래스의 hash 메서드
public static int hash(Object... values) {
    return Arrays.hashCode(values);
}

// Arrays 클래스의 hashCode 메서드
public static int hashCode(Object a[]) {
    if (a == null)
        return 0;

    int result = 1;
```

```
    // 이러한 수식에 의해서 객체의 필드값이 다르면 "가급적" 서로 다른 값의 정수를 반환할 수 있게 됨
    for (Object element : a)
        result = 31 * result + (element == null ? 0 : element.hashCode());

    return result;
}
```

hashCode와 equals의 메서드의 관계

- 1번 법칙) 만약 **두 객체의 equals 반환값이 참(즉, 논리적으로 같은 객체)이라면 두 객체의 해시코드 값은 반드시 같아야** 함
- 2번 법칙) 비록 두 객체가 hashCode 메서드를 통해 똑같은 해시코드 값을 반환한다 해도, **해시값 충돌(=hash collision)**이 일어날 수 있기 때문에 두 객체가 논리적으로 같은 객체(equals 반환값이 참)라는 보장은 할 수 없음

```
public static void main(String[] args) {
    Student s1 = new Student("A1234", "김미림", 19, "여성", "게임");
    Student s2 = new Student("A1234", "김미림", 19, "여성", "게임");
    Student s3 = new Student("A2345", "박미림", 30, "남성", "낚시");

    System.out.println(s1.equals(s2)); // true
    System.out.println(s1.equals(s3)); // false
    System.out.println(s1.hashCode());
    System.out.println(s2.hashCode()); // s1과 같음 (equals 메서드가 참이면 해시코드는 반드시 같다는 요건 충족)
    System.out.println(s3.hashCode()); // s1과 다름 (하지만 "같을수도" 있음)

    Set<Student> set = new HashSet<>();
    set.add(s1);
    set.add(s2);
    set.add(s3);
    System.out.println(set.size()); // 2 (하지만 hashCode 메서드 구현을 주석처리하면?)

    Map<Student, String> map = new HashMap<Student, String>();
    map.put(s1, "Hello");
    map.put(s2, "World"); // "Hello"를 덮어써서 World가 됨
    map.put(s3, "Java");
    System.out.println(map.size()); // 2 (하지만 hashCode 메서드 구현을 주석처리하면?)
    System.out.println(map.get(s2)); // "World"
    System.out.println(map.get(s1)); // 똑같이 "World" (하지만 hashCode 메서드 구현을 주석처리하면?)
}
```

- hashCode 메서드의 오버라이딩 여부에 따라 서로 다른 결과가 나오는 이유는 내부적으로 속도를 향상시키기 위해서 **실제 논리적 비교 메서드인 equals 메서드를 호출하기 전 hashCode를 이용해서 논리적 동등 비교 작업을 대신**하기 때문임
    - 내부적으로 논리적 동등 비교 작업을 진행하고자 할 때, 먼저 hashCode 값을 비교하고 **그 값이 다르다면 equals 메서드를 호출할 필요가 없어짐** (1번 법칙 생각해보기)
    - 하지만 **해시코드 값이 같다고 해서 "반드시" 논리적으로 동등하다고 볼 수 없으므로** 이 경우에는 equals 메서드까지 호출하여 실제로 논리적으로 동등한지 비교 함 (2번 법칙 생각해보

기)

- 이름이 Hash로 시작하는 자료구조 콜렉션 클래스뿐만 아니라 다양한 클래스에서 hashCode 메서드를 활용하므로 반드시 equals 메서드를 오버라이드했다면 hashCode 메서드도 오버라이드해야 함
    - 그리고 높은 확률로 equals 메서드는 오버라이드하게 될 것이므로 서로 쌍(equals, hashCode 둘 다)을 이뤄서 오버라이딩하도록 되어있음
- http://tutorials.jenkov.com/java-collections/hashcode-equals.html
- https://www.baeldung.com/java-hashcode

toString

- toString 메소드는 해당 객체 인스턴스에 대한 유용한 정보를 문자열로 반환하는 역할을 함
- 반환되는 문자열은 클래스 이름과 함께 구분자로 @가 사용되며, 그 뒤로 16진수 해시 코드(hash code)가 추가되는 형식으로 기본 구현이 제공됨

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

- 기본 구현을 사용하면 유용한 정보를 얻기 힘들기 때문에 보통 toString 메서드를 오버라이드하여 유용한 정보를 출력하도록 재정의함
    - 여기서 유용한 정보는 일반적으로 **클래스에 포함된 모든 필드 정보**임

자동 생성한 toString 메서드 구현 살펴보기

```
// Student 클래스의 모든 필드값을 출력할 수 있도록 오버라이딩
@Override
public String toString() {
    return "Student{" +
            "id='" + id + '\'' +
            ", name='" + name + '\'' +
            ", age=" + age +
            ", gender='" + gender + '\'' +
            ", hobby='" + hobby + '\'' +
            '}';
}
```

- 자동 구현된 메서드의 내용은 **사용하는 IDE 및 버전마다 상이**할 수 있음을 유의

(*) clone

- 깊은 복사(deep copy) 작업을 직접 구현하고자 할 때 오버라이드하는 메서드
    - 타언어의 복사 생성자처럼 작동하는 메서드

```
class Person {
    public String id;
    public String name;
    public int age;

    public Person(String id, String name, int age) {
```

```
        this.id = id;
        this.name = name;
        this.age = age;
    }
}

class CloneableObject implements Cloneable {
    // 원시 타입 값
    public int x;
    public int y;

    // 객체 타입 값
    public Person p;
    // 배열도 객체이므로 객체 타입으로 취급
    // https://stackoverflow.com/questions/8781022/is-an-array-an-object-in-java
    public int[] arr = { 1, 2, 3 };

    public CloneableObject() {
        x = 100;
        y = 200;
        p = new Person("1234", "dummy", 100);
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        System.out.println("call override clone method!");

        // 원시 타입 값인 x, y를 복사하기 위해서 기본 clone 메서드 호출
        CloneableObject c = (CloneableObject) super.clone();

        // 코드를 줄이기 위해서 Person의 모든 필드를 public으로 선언하였으나 실제 상황이라면 게터, 세터 사용하여 값 얻어오도록 해야 할 것!
        // 새 Person 인스턴스를 생성하고 값을 복사 (deep copy 진행)
        c.p = new Person(this.p.id, this.p.name, this.p.age);

        // 새로 배열 생성 후 내용 모두 복사
        c.arr = new int[this.arr.length];
        int idx = 0;
        for(int i : this.arr) {
            c.arr[idx] = i;
            idx++;
        }

        return c;
    }
}
```

Wrapper 클래스

- 자바에서 제공하는 원시형(primitive) 타입 8개
    - byte, short, int, long, float, double, char, boolean
- 8개의 원시형 타입에 해당하는 데이터를 **객체로 포장해 주는 클래스를 래퍼 클래스(Wrapper class)**라고 부름

    - 일반적으로 앞 글자를 대문자로 바꾸면 래퍼 클래스의 이름이 되지만 예외가 있음 (예외 : int -> Integer, char -> Character)
- 래퍼 클래스는 각각의 타입에 해당하는 데이터를 인수로 전달받아서 해당 값을 가지는 객체로 만듦
- 래퍼 클래스는 모두 java.lang 패키지에 포함되어 제공됨
- 보통 기본 타입의 데이터를 **객체로 취급해야 하는 경우**에 사용함

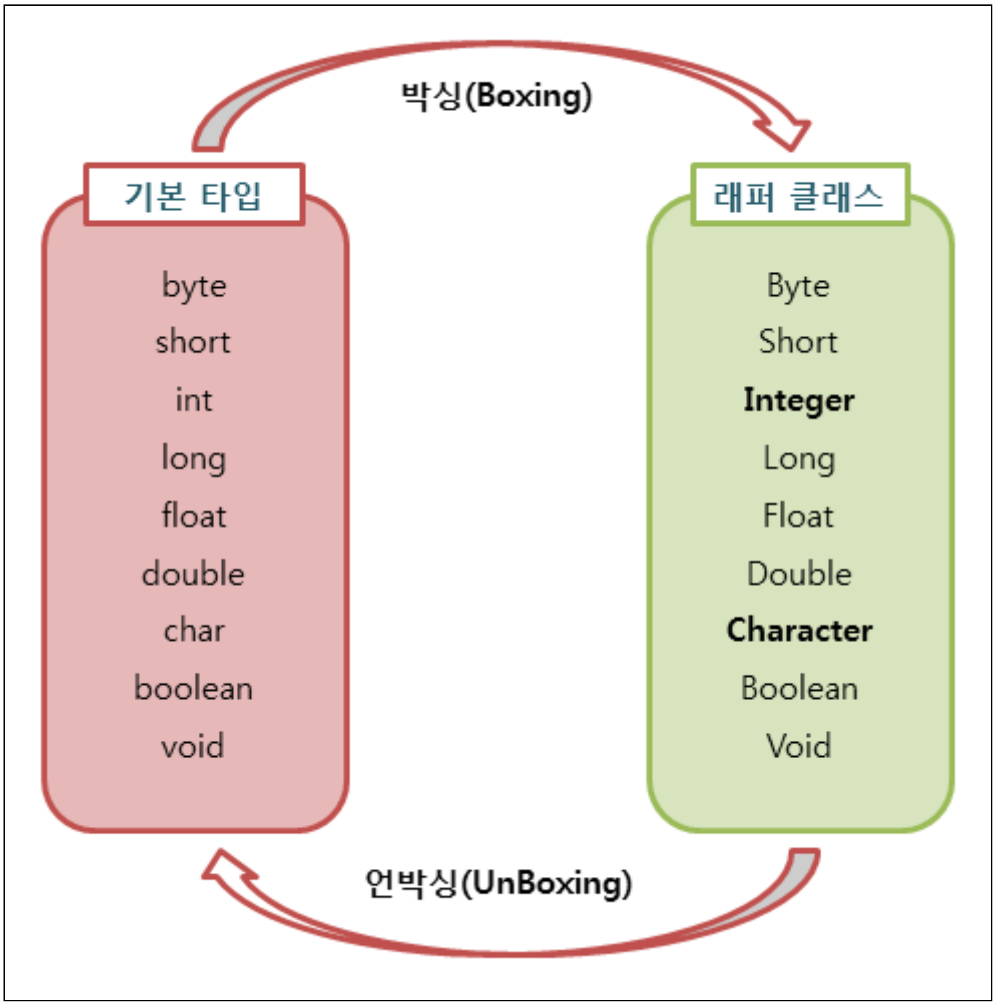
기본 타입	래퍼 클래스
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Wrapper 클래스 생성 코드

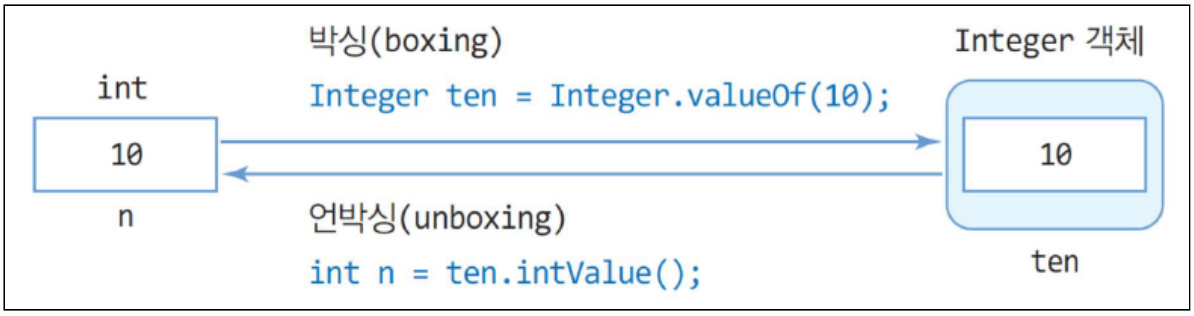
```
// Wrapper 클래스의 생성자 호출하며 원시형 값을 전달
Integer iw = new Integer(1);
Double dw = new Double(1.234);

// 혹은 오토박싱을 이용하여 객체 생성 가능
Integer iw2 = 1;
Double dw2 = 1.234;
```

박싱과 언박싱



- 그림과 같이 기본 타입의 데이터를 래퍼 클래스의 인스턴스로 변환하는 과정을 박싱(boxing)이라고 함
- 래퍼 클래스의 객체에 저장된 값을 다시 기본 타입의 데이터로 꺼내는 과정을 언박싱(unboxing)이라고 함



- 오토 박싱(auto boxing)과 오토 언박싱(auto unboxing)
  - JDK 1.5부터는 박싱과 언박싱이 필요한 상황에서 자바 컴파일러가 이를 자동으로 처리해 줍니다. 이렇게 자동화된 박싱과 언박싱을 오토 박싱(AutoBoxing)과 오토 언박싱(AutoUnBoxing)이라고 부릅니다.

박싱, 언박싱 코드

```
// 박싱
Integer num = new Integer(17);
// 언박싱
int n = num.intValue();
```

오토박싱, 오토언박싱 코드

```java
// 원래는
// Character ch = new Character('X');
// 라고 써야 하는데 "오토박싱"에 의해서 아래와 같이 써줘도 됨
Character ch = 'X';

// 원래는
// char c = ch.charValue();
// 라고 써야 하는데 "오토언박싱"에 의해서 아래와 같이 써줘도 됨
char c = ch;
```

Wrapper 클래스 관련 유용한 메서드

- parse로 시작하는 값 변환 메서드와 16진수 변환 메서드 등을 활용 가능

```java
int v1 = Integer.parseInt("100");
double v2 = Double.parseDouble("1.234");
boolean v3 = Boolean.parseBoolean("true");

// 정수값을 16진수로 변환
System.out.println(Integer.toHexString(255)); // "ff"

// 정수값을 2진수로 변환
System.out.println(Integer.toBinaryString(255)); // "11111111"
// 숫자에 포함된 비트1 개수 반환 => 255이므로 8을 반환
System.out.println(Integer.bitCount(255));
```

Wrapper 타입의 값 비교시 주의점

```java
// 래퍼 클래스 사용하면서 값을 비교할 때 주의점
Integer i1 = new Integer(100);
Integer i2 = new Integer(100);
// 객체 타입의 값 비교이므로 equals 메서드를 사용해야 함
System.out.println("i1.equals(i2) : " + i1.equals(i2)); // true
// == 연산자 사용시 리퍼런스 비교를 진행하므로 false
System.out.println("i1 == i2 : " + (i1 == i2));

Boolean wb1 = new Boolean(true);
Boolean wb2 = new Boolean(true);
System.out.println("wb1.equals(wb2) : " + wb1.equals(wb2)); // true
System.out.println("wb1 == wb2 : " + (wb1 == wb2)); // false (주소 비교)
```

String, StringBuffer, StringTokenizer 클래스

String 클래스

- 자바 String과 관련된 상식

```
1. String은 불변(immutable) 값
2. String 객체 내부에 char 배열을 이용하여 글자를 저장 (Null-terminated string은 아님)
3. JVM 내부에서 텍스트를 표현할 때 UTF-16 방식을 사용함 (따라서 char 자료형은 2바이트)
```

- 자바에서 사용하는 리터럴(literal) 문자열들(new String으로 생성하지 않는 모든 문자열)은 모두 **String Pool에 저장**됨 (일종의 fly-weight design pattern)
  - 따라서 메모리를 아끼기 위해서 **같은 내용의 문자열의 경우 String Pool의 같은 리퍼런스를 가리키게 함**
  - 단, new 키워드를 이용하여 새 문자열을 동적 생성할 경우 이는 일반적인 Heap 메모리에 저장됨
    - 이때 intern() 메소드를 사용하여 새로 동적 생성된 문자열을 String Pool에 저장 가능

```
public static void main(String[] args) {
    // "Hello"와 "World" 문자열은 모두 리터럴 문자열이므로 String Constant Pool에 생성됨
    String s1 = "Hello";
    s1 = " World!"; // 대입은 가능 하지만 SCP 메모리 영역에 존재하는 "Hello"라는 문자열 값 자체가 변경되지는 않음 (불변)

    // 어차피 컴파일러 입장에서는 컴파일 과정에 최종 결과물이 "abc"라는 것을 알기 때문에 역시 String Pool에 "abc" 문자열을 만들어 같은 리퍼런스를 가리키게 함
    String cs1 = "ab" + "c";
    String cs2 = "a" + "bc";
    System.out.println("cs1 == cs2 : " + (cs1 == cs2));

    // 정리
    // true (둘 다 String Constant Pool에 생성됨)
    System.out.println("\"ZetCode\" == \"ZetCode\" : " + ("ZetCode" == "ZetCode"));
    // false (new 키워드를 이용하여 동적 생성한 문자열은 Heap 메모리에 저장됨)
    System.out.println("\"ZetCode\" == new String(\"ZetCode\") : " + ("ZetCode" == new String("ZetCode")));
    // true (문자열 리터럴을 더해주는 것이므로 왼쪽, 오른쪽 모두 String Pool에 존재하는 "ZetCode"를 가리키게 됨)
    System.out.println("\"ZetCode\" == \"Zet\" + \"Code\" : " + ("ZetCode" == "Zet" + "Code"));
    // true (intern() 메소드 호출로 인하여 String Constant Pool에 존재하는 리퍼런스를 가리키게 됨)
    System.out.println("\"ZetCode\" == new String(\"ZetCode\").intern() : " + ("ZetCode" == new String("ZetCode").intern()));
    // false (trim 내부에서 substring 메소드를 호출하고 해당 메소드 내부에서 새로운 문자열 생성(new String))
    System.out.println("\"ZetCode\" == \" ZetCode \".trim() : " + ("ZetCode" == " ZetCode ".trim()));

    // 또 다른 intern 메소드 예제
    // 문자열을 모두 힙 영역에 생성
    String interned1 = new String("Java Programming");
    String interned2 = new String("Java Programming");
    System.out.println("without calling intern method : " + (interned1 == interned2)); // false
```

```
    interned1 = interned1.intern();
    System.out.println("after calling intern method once : " + (interned1 ==
interned2)); // false
    interned2 = interned2.intern();
    System.out.println("after calling intern method both : " + (interned1 ==
interned2)); // true

    // 팁) 문자열을 생성시 new String을 쓰지 말고 문자열 리터럴을 활용할 것
    String s2 = " Hello ";
    System.out.println(s2);

    // charAt : 특정 인덱스의 문자 반환 (0번 인덱스가 첫 글자)
    System.out.println(s2.charAt(1) == 'H');

    // indexOf : 반대로 첫 번째로 만나는 특정 글자의 인덱스를 반환
    System.out.println(s2.indexOf('H') == 1);
    // l이 두 개이지만 처음 만나는 l의 위치(3)만 반환
    System.out.println(s2.indexOf('l') == 3);
    // 루프 이용하여 모든 'l' 글자의 위치를 출력하는 코드
    for (int index = s2.indexOf('l'); index >= 0; index = s2.indexOf('l', index
+ 1)) {
        System.out.println("index : " + index);
    }

    // trim : 앞, 뒤의 공백 자르기
    s2 = s2.trim();
    System.out.println("after s2.trim() : " + s2);

    // substring : 문자열 일부분 자르기 (begin index, end index)
    // Hello의 0번째 index 글자는 e, 4번째 글자는 l 따라서 결과는 Hell
    System.out.println("s2.substring(0, 4) : " + s2.substring(0, 4));

    s1 = "1234";
    // http://stackoverflow.com/questions/508665/difference-between-parseint-and-
valueof-in-java
    // valueOf는 래퍼 클래스의 인스턴스를 반환 (교과서 116쪽)
    System.out.println("Integer.valueOf(s1) : " + Integer.valueOf(s1));
    // Integer -> parseInt, Float -> parseFloat, Double -> parseDouble 등등 여러
변환 메소드가 존재 (이 메소드들은 원시(primitive)값을 반환)
    System.out.println("Integer.parseInt(s1) : " + Integer.parseInt(s1));
    s1 = "1234.5678";
    System.out.println("Float.valueOf(s1) : " + Float.valueOf(s1)); //
1234.5677?

    // 문자열 비교 (객체 타입의 경우 '내용'을 비교할 때 ==가 아닌 equals 메소드를 사용,
==는 주소 비교에 사용)
    s1 = "Hello";
    s2 = "Hello";

    // 같은 메모리 주소 공유 (컴파일시에 같은 문자열임을 알 수 있으므로 메모리 주소를 구분
할 필요가 없고, 따라서 같은 주소를 대입! (String Pool))
    System.out.println("s1 == s2 : " + (s1 == s2)); // true
    // 강제로 새 메모리 영역에 String 객체 생성 (new 키워드 사용, 일반 Heap 영역에 저장되
고 String Pool에는 저장되지 않음)
    s2 = new String("Hello");
    // 여기서 ==는 같은 객체인지를 비교(reference equality)하기 때문에 같은 값을 가지고
있음에도 불구하고 비교 결과는 false가 됨
    System.out.println("s1 == s2 : " + (s1 == s2)); // false
```

```
    // *문자열 비교는 equals 메소드를 사용 (값을 비교)
    System.out.println("s1.equals(s2) : " + (s1.equals(s2)));

    // 추가적으로 알면 좋은 메소드
    // http://stackoverflow.com/questions/47605/string-concatenation-concat-vs-operator
    String s3 = s1.concat(" World");
    System.out.println("s3 : " + s3);
    // startsWith : prefix 검사
    System.out.println("s3.startsWith(\"Hello\") : " + s3.startsWith("Hello"));
    // endsWith : suffix 검사
    System.out.println("s3.endsWith(\"World\") : " + s3.endsWith("World"));
    // replace : replace(char oldChar, char newChar) => 등장하는 oldChar를 newChar로 변경 (밑의 코드는 등장하는 모든 l을 i로 변경)
    System.out.println("s3.replace('l', 'i') : " + s3.replace('l', 'i'));
    // split : 특정 정규식 문자열을 기준으로 텍스트 분할하여 배열에 저장
    String s4 = "010-1234-5678";
    // "-" 글자를 기준으로 자르기
    String[] ret = s4.split("-");
    System.out.println("s4.split(\"-\") length : " + ret.length);
    for(String s : ret) {
        System.out.println(s);
    }
    // format : C 언어의 printf 함수랑 비슷한 역할을 하며 구조화된 문자열을 만드는데 사용됨
    // http://nukestorm.tistory.com/51
    System.out.println(String.format("%s %d %f", "Hello", 1234, 3.14));
}
```

StringBuffer 클래스

- 불변인 String 클래스와는 다르게 문자열을 조작하는 연산을 지원할 수 있도록 만든 "가변 문자열" 클래스인 StringBuffer가 제공됨
    - 문자열을 변경하는 작업을 많이 진행해야 하는 상황이라면 String 클래스 대신 StringBuffer 클래스를 사용
- 똑같은 역할을 하는 StringBuilder 클래스가 있지만 해당 클래스는 스레스안전(thread-safe)하지 않음, 만약 싱글 스레드에서만 접근하는것을 보장할 수 있으면 성능상의 이득을 취하기 위해서 StringBuilder 클래스를 사용하는 것이 권장됨

```
public class StringBufferClassExample {
    public static void main(String[] args) {
        // StringBuffer 클래스
        StringBuffer sBuffer = new StringBuffer();

        // capacity의 양 설정 (초기 char 배열의 크기)
        /*
        Every string buffer has a capacity. As long as the length of the character sequence contained in
        the string buffer does not exceed the capacity, it is not necessary to allocate a new internal
        buffer array. If the internal buffer overflows, it is automatically made larger.
        */
```

```
        // 적절한 정도의 capacity를 설정하면 새 메모리 할당(memory allocation)이 일어나지 않아 빠른 문자열 이어붙이기 및 조작이 가능
        sBuffer.ensureCapacity(100);
        // capacity가 100이므로 문자열 길이가 100을 넘어가는 시점부터 메모리 할당이 일어날 수 있음
        System.out.println(sBuffer.capacity());

        // 현재 length는 0
        System.out.println(sBuffer.length());

        // append 메소드를 이용하여 내용 추가 가능
        sBuffer.append("Hello");
        sBuffer.append(" World!");

        // insert 메소드를 이용하여 필요한 위치에 문자열 삽입 가능
        sBuffer.insert(0, "0Insert ");
        System.out.println(sBuffer);

        // 앞에서 삽입한 문자열 바로 뒤에 삽입
        sBuffer.insert(8, "8Insert ");
        System.out.println(sBuffer);

        // delete 메소드를 이용하여 특정 시작 위치부터 끝 위치까지의 문자열을 삭제 가능
        sBuffer.delete(8, 16);

        // println내부에서 object.toString을 호출하여 String 객체를 출력
        System.out.println(sBuffer);
        System.out.println(sBuffer.length());

        // StringBuffer에서 String 객체 생성
        // (new 연산자를 통하여 Heap에 메모리 할당이 일어나므로 반복문 내부에서 사용은 권장되지 않음)
        String res = sBuffer.toString();
        System.out.println(res);
    }
}
```

StringTokenizer 클래스

```
public class StringTokenizerClassExample {
    public static void main(String args[]) {
        System.out.println("StringTokenizer Example");

        String s = "Hello World";
        // 기본 구분 문자(delimiter)는 스페이스
        StringTokenizer st = new StringTokenizer(s);
        System.out.println("st token count : " + st.countTokens());
        while(st.hasMoreTokens()) {
            String next = st.nextToken();
            System.out.println(next);
        }
        // 한 번 토큰을 읽어온 다음 토큰 개수를 count하면 0이 뜸 (한 번 더 순회해야 한다면 다시 StringTokenizer 객체를 생성)
        System.out.println("st token count (after iteration) : " +
st.countTokens());
```

```
        s = "010-1234-5678";
        // 구분 문자를 '-'로 지정하여 StringTokenizer 생성
        StringTokenizer st2 = new StringTokenizer(s, "-");
        System.out.println("st2 token count : " + st2.countTokens());
        while(st2.hasMoreTokens()) {
            String next = st2.nextToken();
            System.out.println(next);
        }
    }
}
```

Math 클래스

- 수학과 관련된 다양한 메서드 제공

```
public class MathClassExample {
    public static int getRandomNumberFromRange(int min, int max) {
        return min + (int)(Math.random() * ((max - min) + 1));
    }

    public static void main(String args[]) {
        System.out.println("Math Class Example");

        /* 상수 */
        System.out.println(Math.E); // 자연대수
        System.out.println(Math.PI); // PI 값

        /* 대표 메소드 */
        // 절대값 함수
        System.out.println("Math.abs(-100) : " + Math.abs(-100));

        // 반올림 함수
        // 실제 적용되는 공식 (long) Math.floor(x + 0.5d)
        // 정수값(long)을 리턴
        System.out.println("Math.round(0.4) : " + Math.round(0.4)); // 0
        System.out.println("Math.round(0.5) : " + Math.round(0.5)); // 1
        System.out.println("Math.round(0.9) : " + Math.round(0.9)); // 1

        /*
        // 음수값 반올림시 헷갈리지 않도록 주의
        System.out.println("Math.round(-0.4) : " + Math.round(-0.4)); // 0
        System.out.println("Math.round(-0.5) : " + Math.round(-0.5)); // 0
        System.out.println("Math.round(-0.9) : " + Math.round(-0.9)); // -1
        */

        // 제곱(power) 함수 (값, 제곱값)
        System.out.println("Math.pow(2, 2) : " + Math.pow(2, 2)); // 4
        System.out.println("Math.pow(2, 3) : " + Math.pow(2, 3)); // 8

        // 제곱근(square root) 함수
        System.out.println("Math.sqrt(4) : " + Math.sqrt(4)); // 2 (루트 4)
        System.out.println("Math.sqrt(2) : " + Math.sqrt(2)); // 1.414... (루트
2)
```

```
        // random 메소드 사용법 (내부적으로는 Random 객체를 사용함)
        // Returns a double value with a positive sign, greater than or equal to
0.0 and less than 1.0.
        System.out.println("Math.random() : " + Math.random());
        // 0 ~ 9까지 랜덤 숫자 구하기 (정수로 캐스팅하는 부분 잘 이해해보기)
        int r1 = (int) (Math.random() * 10);
        System.out.println("r1 : " + r1);
        // 1 ~ 10까지 랜덤 숫자 구하기 (0.0 ~ 9.9999.. 까지 결과값이 나오고 이를 정수로
만들면 결과값의 범위는 0~9, 거기에 1을 더함)
        int r2 = ((int) (Math.random() * 10)) + 1;
        System.out.println("r2 : " + r2);
        // min ~ max까지 랜덤 숫자 구하기
        System.out.println("getRandomNumberFromRange(100, 200) : " +
getRandomNumberFromRange(100, 200));

        /* 삼각 함수 */
        // sin (cos, tan도 있음, 인자는 모두 라디안 값임을 주의!)
        System.out.println("Math.sin(0) : " + Math.sin(0));
        System.out.println("Math.sin(90) : " + Math.sin(90)); // 0.8939...
        // toRadians 함수 이용하여 도(degree) -> 라디안(radian) 변환
        System.out.println("Math.sin(Math.toRadians(90)) : " +
Math.sin(Math.toRadians(90))); // 1

        // 올림(ceiling) 함수
        System.out.println("Math.ceil(0.1) : " + Math.ceil(0.1));

        // 내림 함수
        System.out.println("Math.floor(0.9) : " + Math.floor(0.9));

        // 최소 값
        System.out.println("Math.min(1, 3) : " + Math.min(1, 3));

        // 최대 값
        System.out.println("Math.max(1, 3) : " + Math.max(1, 3));
    }
}
```

Calendar 클래스

- 날짜와 시간 정보 관련된 작업을 진행하는 클래스

```
public class CalendarClassExample {
    public static void main(String args[]) {
        Date now = new Date();
        // 추상 클래스 Calendar 사용하여 날짜 및 시간 받아오기 가능
        // Calendar의 싱글턴 객체를 받아오기 (현재 시간을 기준으로 캘린더 객체가 생성됨)
        Calendar c = Calendar.getInstance();
        // 먼저 날짜 객체를 이용하여 캘린더 인스턴스의 날짜를 설정 가능
        c.setTime(now);
        // 날짜 및 시간 출력
        System.out.println("YEAR : " + c.get(Calendar.YEAR));
    // 년
        System.out.println("MONTH : " + (c.get(Calendar.MONTH) + 1));
    // 월 (월의 경우 리턴값의 범위는 0~11 따라서 1을 더해줘야 함)
```

```
        System.out.println("DAY_OF_MONTH : " + c.get(Calendar.DAY_OF_MONTH));
    // 한 달 중의 날짜
        System.out.println("HOUR : " + c.get(Calendar.HOUR));
    // 시
        System.out.println("MINUTE : " + c.get(Calendar.MINUTE));
    // 분
        System.out.println("SECOND : " + c.get(Calendar.SECOND));
    // 초
        System.out.println("MILLISECOND : " + c.get(Calendar.MILLISECOND));
    // 밀리초
        System.out.println("HOUR_OF_DAY : " + c.get(Calendar.HOUR_OF_DAY));
    // 24시간제를 기준으로 한 시간
        System.out.println("DAY_OF_WEEK : " + c.get(Calendar.DAY_OF_WEEK));
    // 한 주의 요일 (일요일부터 1, 토요일이 7)
        System.out.println("AM_PM : " + (c.get(Calendar.AM_PM) == Calendar.AM));
   // AM_PM => 오전, 오후 여부(AM, PM 상수와 비교)
    }
}
```

LocalDate, LocalDateTime 클래스

- java.time 패키지 소속의 클래스로 1.8버전부터 추가된 표준 날짜, 시간용 클래스
    - 기존의 Date 클래스 대신 사용이 권장됨