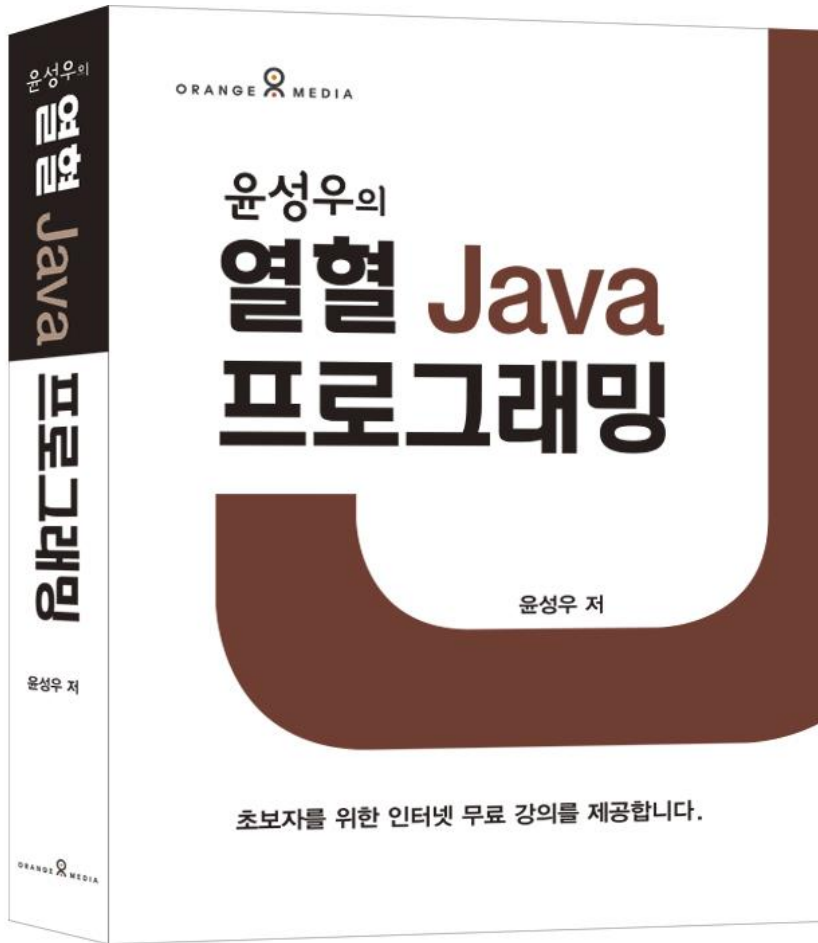


2022년 봄학기

---

JAVA

나사렛대학교  
IT융합학부  
김광기



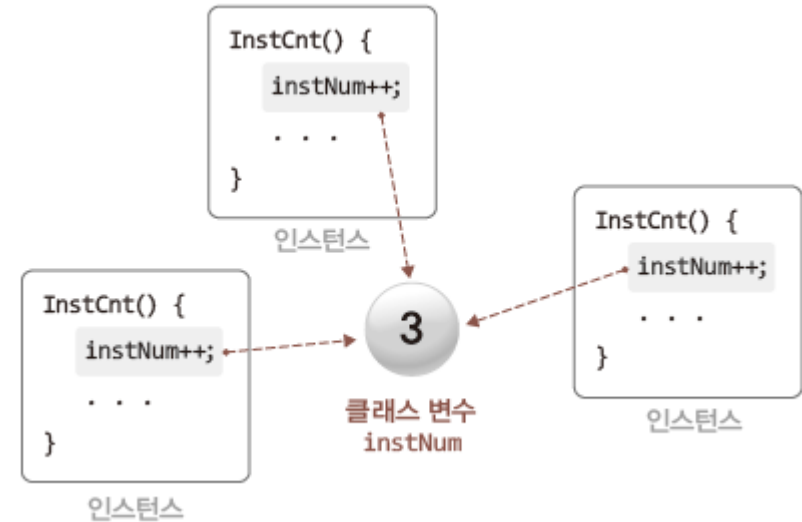
# 열혈 Java 프로그래밍

Chapter 10. 클래스 변수와 클래스 메소드

## 10-1. static 선언을 붙여서 선언하는 클래스 변수

# 선언된 클래스의 모든 인스턴스가 공유하는 클래스 변수

```
class InstCnt {  
    static int instNum = 0;    // 클래스 변수 (static 변수)  
  
    InstCnt() {  
        instNum++;  
        System.out.println("인스턴스 생성: " + instNum);  
    }  
}  
  
class ClassVar {  
    public static void main(String[] args) {  
        InstCnt cnt1 = new InstCnt();  
        InstCnt cnt2 = new InstCnt();  
        InstCnt cnt3 = new InstCnt();  
    }  
}
```



```
명령 프롬프트
C:\JavaStudy>java ClassVar
인스턴스 생성: 1
인스턴스 생성: 2
인스턴스 생성: 3
C:\JavaStudy>
```

# 클래스 변수의 접근 방법

---

## 클래스 내부 접근

- static 변수가 선언된 클래스 내에서는 이름만으로 직접 접근 가능

## 클래스 외부 접근

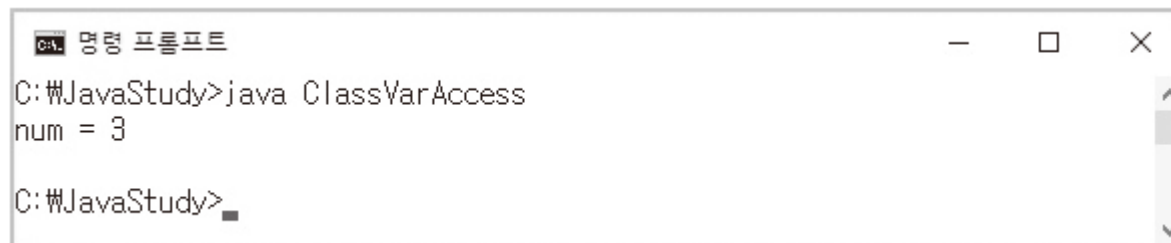
- private으로 선언되지 않으면 클래스 외부에서도 접근 가능
- 접근 수준 지시자가 허용하는 범위에서 접근 가능
- 클래스 또는 인스턴스의 이름을 통해 접근

# 클래스 변수 접근의 예

```
class AccessWay {
    static int num = 0;

    AccessWay() { incrCnt(); }
    void incrCnt() {
        num++; // 클래스 내부에서 이름을 통한 접근
    }
}

class ClassVarAccess {
    public static void main(String[] args) {
        AccessWay way = new AccessWay();
        way.num++; // 외부에서 인스턴스의 이름을 통한 접근
        AccessWay.num++; // 외부에서 클래스의 이름을 통한 접근
        System.out.println("num = " + AccessWay.num);
    }
}
```



```
명령 프롬프트
C:\JavaStudy>java ClassVarAccess
num = 3
C:\JavaStudy>
```

# 클래스 변수의 초기화 시점과 초기화 방법

---

```
class InstCnt {  
    static int instNum = 100;  
    클래스 변수의 적절한 초기화 위치  
    InstCnt() {  
        instNum++;  
        System.out.println("인스턴스 생성: " + instNum);  
    }  
}
```

클래스 변수는 생성자 기반 초기화 하면 안된다!

이 경우 인스턴스 생성시마다 값이 리셋!

```
class OnlyClassNoInstance {  
    public static void main(String[] args) {  
        InstCnt.instNum -= 15;    // 인스턴스 생성 없이 instNum에 접근  
        System.out.println(InstCnt.instNum);  
    }  
}
```

# 클래스 변수의 활용의 예

---

```
class Circle {  
    static final double PI = 3.1415;  
    private double radius;  
  
    Circle(double rad) {  
        radius = rad;  
    }  
    void showPerimeter() {  
        double peri = (radius * 2) * PI;  
        System.out.println("둘레: " + peri);  
    }  
    void showArea() {  
        double area = (radius * radius) * PI;  
        System.out.println("넓이: " + area);  
    }  
}
```

인스턴스 별로 가지고 있을 필요가 없는 변수

- 값의 참조가 목적인 변수
- 값의 공유가 목적인 변수

그리고 그 값이 외부에서도 참조하는 값이라면 public으로 선언한다.



## 10-2. static 선언을 붙여서 정의 하는 클래스 메소드

# 클래스 메소드의 정의와 호출

---

```
class NumberPrinter {  
    private int myNum = 0;  
    static void showInt(int n) { System.out.println(n); }  
    static void showDouble(double n) {System.out.println(n); }  
  
    void setMyNumber(int n) { myNum = n; }  
    void showMyNumber() { showInt(myNum); }  
}  
                        내부 접근
```

클래스 메소드의 성격 및 접근 방법이  
클래스 변수와 동일하다.

```
class ClassMethod {  
    public static void main(String[] args) {  
외부 접근 NumberPrinter.showInt(20);  
        NumberPrinter np = new NumberPrinter();  
외부 접근 np.showDouble(3.15);  
        np.setMyNumber(75);  
        np.showMyNumber();  
    }  
}
```

## 클래스 메소드로 정의하는 것이 옳은 경우

---

```
class SimpleCalculator {  
    static final double PI = 3.1415;  
  
    static double add(double n1, double n2) {  
        return n1 + n2;  
    }  
    static double min(double n1, double n2) {  
        return n1 - n2;  
    }  
    static double calcCircleArea(double r) {  
        return PI * r * r;  
    }  
    static double calcCirclePeri(double r) {  
        return PI * (r * 2);  
    }  
}
```

단순 기능 제공이 목적인 메소드들, 인스턴스 변수와 관련 지을 이유가 없는 메소드들은 static으로 선언하는 것이 옳다.

# 클래스 메소드에서 인스턴스 변수에 접근이 가능할까?

---

```
class AAA {  
    int num = 0;  
    static void addNum(int n) {  
        num += n;  
    }  
}
```

논리적으로 이 문장이 유효할 수 있는지를 생각해보자.

10-3. System.out.println 그리고

```
public static void main()
```

## System.out.println()에서 out과 println의 정체는?

---

```
java.lang.System.out.println(...);
```

| System은 java.lang 패키지에 묶여 있는 클래스의 이름

| 그러나 컴파일러가 다음 문장을 삽입해 주므로 java.lang을 생략할 수 있다.

```
| import java.lang.*;
```

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

| out은 클래스 System의 이름을 통해 접근하므로,

| 이는 System 클래스의 클래스 변수 이름임을 유추할 수 있다.

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

| println은 out이 참조하는 인스턴스의 메소드이다.

# main 메소드가 public이고 static인 이유는?

---

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

| static인 이유! 인스턴스 생성과 관계없이 제일 먼저 호출되는 메소드이다.

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

| public인 이유! main 메소드의 호출 명령은 외부로부터 시작되는 명령이다.  
| 단순히 일종의 약속으로 이해해도 괜찮다.

## main 메소드를 어디에 위치시킬 것인가?

---

```
class Car {  
    void myCar() {  
        System.out.println("This is my car");  
    }  
}
```

```
public static void main(String[] args) {  
    Car c = new Car();  
    c.myCar();  
    Boat t = new Boat();  
    t.myBoat();  
}
```

```
}
```

Boat 클래스로 이동시킨다면 달라지는 것은?

```
class Boat {  
    void myBoat() {  
        System.out.println("This is my boat");  
    }  
}
```




## 10-4. 또 다른 용도의 static 선언

# static 초기화 블록

---

```
class DateOfExecution {  
    static String date;    // 프로그램의 실행 날짜를 저장하기 위한 변수  
  
    public static void main(String[] args) {  
        System.out.println(date);  
    }  
}
```



```
static {  
    LocalDate nDate = LocalDate.now();  
    date = nDate.toString();  
}
```

인스턴스 생성과 관계 없이 static 변수가 메모리 공간에 할당될 때 실행이 된다.

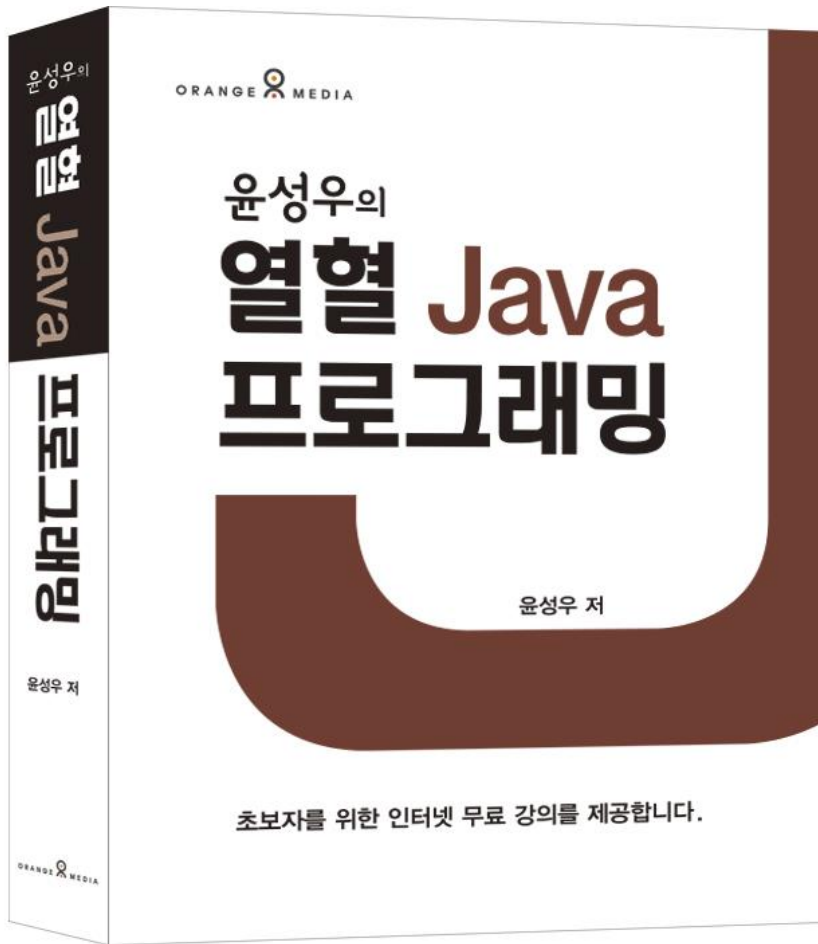
# static import 선언

---

```
System.out.println(Math.PI);  
                    java.lang.Math.PI
```

```
System.out.println(PI);  
                    import static java.lang.Math.PI;
```

추가된 문법이긴 하지만...



# 열혈 Java 프로그래밍

Chapter 11. 메소드 오버로딩과 String 클래스

## 11-1. 메소드 오버로딩

# 메소드 오버로딩

---

호출된 메소드를 찾을 때 참조하게 되는 두 가지 정보

- 메소드의 이름
- 메소드의 매개변수 정보

따라서 이 둘 중 하나의 형태가 다른 메소드를 정의하는 것이 가능하다.

```
class MyHome {  
    void mySimpleRoom(int n) {...}  
    void mySimpleRoom(int n1, int n2) {...}  
    void mySimpleRoom(double d1, double d2) {...}  
}
```

} 메소드 오버로딩

# 메소드 오버로딩의 예

---

```
void simpleMethod(int n) {...}  
void simpleMethod(int n1, int n2) {...}
```

매개변수의 수가 다르므로 성립!

```
void simpleMethod(int n) {...}  
void simpleMethod(double d) {...}
```

매개변수의 형이 다르므로 성립!

```
int simpleMethod() {...}  
double simpleMethod() {...}
```

반환형은 메소드 오버로딩의 조건 아님!

# 오버로딩 관련 피해야할 애매한 상황

---

```
class AAA {  
    void simple(int p1, int p2) {...}  
    void simple(int p1, double p2) {...}  
}
```

다음과 같이 모호한 상황을 연출하지 않는 것이 좋다!

```
AAA inst = new AAA();  
inst. simple(7, 'K');      // 어떤 메소드가 호출될 것인가?
```



# 생성자의 오버로딩

---

```
class Person {  
    private int regiNum;    // 주민등록 번호  
    private int passNum;    // 여권 번호  
  
    Person(int rnum, int pnum) {  
        regiNum = rnum;  
        passNum = pnum;  
    }  
  
    Person(int rnum) {  
        regiNum = rnum;  
        passNum = 0;  
    }  
  
    void showPersonalInfo() {...}  
}
```

```
public static void main(String[] args) {  
    // 여권 있는 사람의 정보를 담은 인스턴스 생성  
    Person jung = new Person(335577, 112233);  
  
    // 여권 없는 사람의 정보를 담은 인스턴스 생성  
    Person hong = new Person(775544);  
  
    jung.showPersonalInfo();  
    hong.showPersonalInfo();  
}
```

생성자의 오버로딩을 통해 생성되는 인스턴스의  
유형을 구분할 수 있다.


ex) 여권이 있는 사람과 없는 사람

ex) 운전 면허증을 보유한 사람과 보유하지 않은 사람

# 키워드 this를 이용한 다른 생성자의 호출

---

```
class Person {  
    private int regiNum;    // 주민등록 번호  
    private int passNum;    // 여권 번호  
  
    Person(int rnum, int pnum) {  
        regiNum = rnum;  
        passNum = pnum;  
    }  
  
    Person(int rnum) {  
        regiNum = rnum;  
        passNum = 0;  
    }  
  
    void showPersonalInfo() {...}  
}
```



```
Person(int rnum) {  
    this(rnum, 0);  
}
```

rnum과 0을 인자로 받는 오버로딩 된 다른 생성자 호출,  
중복된 코드를 줄이는 효과!

# 키워드 `this`를 이용한 인스턴스 변수의 접근

---

```
class SimpleBox {  
    private int data;  
  
    SimpleBox(int data) {  
        this.data = data;  
    }  
}
```

} `this.data`는 어느 위치에서 건 인스턴스 변수 `data`를 의미함

## 11-2. String 클래스

# String 인스턴스 생성의 두 가지 방법

---

```
String str1 = new String("Simple String");
```

```
String str2 = "The Best String";
```

둘 다 String 인스턴스의 생성으로 이어지고 그 결과 인스턴스의 참조 값이 반환된다.

# String 인스턴스와 println 메소드

---

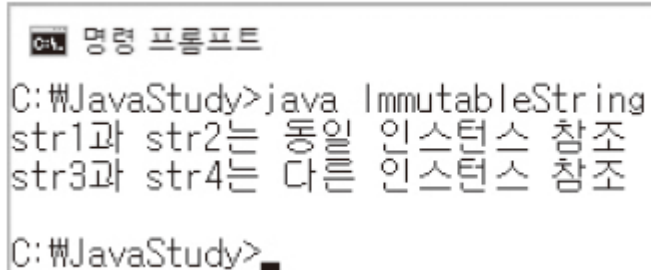
```
public static void main(String[] args) {  
    String str1 = new String("Simple String");  
    String str2 = "The Best String";  
  
    System.out.println(str1);  
    System.out.println(str1.length());  
    System.out.println();    // '개 행'  
  
    System.out.println(str2);  
    System.out.println(str2.length());  
    System.out.println();  
  
    showString("Funny String");  
}  
  
public static void showString(String str) {  
    System.out.println(str);  
    System.out.println(str.length());  
}
```

```
void println() {...}  
void println(int x) {...}  
void println(String x) {...}
```

println 메소드가 다양한 인자를  
전달받을 수 있는 이유는 메소드 오버로딩

# 문자열 생성 방법 두 가지의 차이점

```
class ImmutableString {  
    public static void main(String[] args) {  
        String str1 = "Simple String";  
        String str2 = "Simple String";  
  
        String str3 = new String("Simple String");  
        String str4 = new String("Simple String");  
  
        참조변수의 참조 값 비교  
        if(str1 == str2)  
            System.out.println("str1과 str2는 동일 인스턴스 참조");  
        else  
            System.out.println("str1과 str2는 다른 인스턴스 참조");  
  
        참조변수의 참조 값 비교  
        if(str3 == str4)  
            System.out.println("str3과 str4는 동일 인스턴스 참조");  
        else  
            System.out.println("str3과 str4는 다른 인스턴스 참조");  
    }  
}
```



```
명령 프롬프트  
C:\JavaStudy>java ImmutableString  
str1과 str2는 동일 인스턴스 참조  
str3과 str4는 다른 인스턴스 참조  
C:\JavaStudy>
```

# String 인스턴스는 Immutable 인스턴스

---

String 인스턴스는 **Immutable** 인스턴스!

따라서 생성되는 인스턴스의 수를 **최소화** 한다.

```
public static void main(String[] args) {  
    String str1 = "Simple String";  
    String str2 = str1;  
    . . .
```

```
public static void main(String[] args) {  
    String str1 = "Simple String";  
    String str2 = new String("Simple String");  
    . . .
```

이후로 두 코드에 어떠한 차이점을 부여할 수 있겠는가? (사실상 차이가 없다는 의미)



# String 인스턴스 기반 switch문 구성

---

```
public static void main(String[] args) {  
    String str = "two";  
  
    switch(str) {  
        case "one":  
            System.out.println("one");  
            break;  
        case "two":  
            System.out.println("two");  
            break;  
        default:  
            System.out.println("default");  
    }  
}
```

## 11-3. String 클래스의 메소드

# 문자열 연결시키기

---

```
class StringConcat {  
    public static void main(String[] args) {  
        String st1 = "Coffee";  
        String st2 = "Bread";  
  
        String st3 = st1.concat(st2);  
        System.out.println(st3);  
  
        String st4 = "Fresh".concat(st3);  
        System.out.println(st4);  
    }  
}
```



C:\. 명령 프롬프트

```
C:\#JavaStudy>java StringConcat  
CoffeeBread  
FreshCoffeeBread  
C:\#JavaStudy>
```

# 문자열의 일부 추출

---

```
String str = "ABCDEFGH";
```

```
str.substring(2);
```

a	b	c	d	e	f	G
0	1	2	3	4	5	6

인덱스 2 이후의 내용으로 이뤄진 문자열 "cdefgh" 반환

```
String str = "ABCDEFGH";
```

```
str.substring(2, 4);
```

a	b	c	d	e	f	G
0	1	2	3	4	5	6

인덱스 2 ~ 3에 위치한 내용의 문자열 반환

# 문자열의 내용 비교

```
public static void main(String[] args) {  
    String st1 = "Lexicographically";  
    String st2 = "lexicographically";  
    int cmp;  
  
    if(st1.equals(st2))  
        System.out.println("두 문자열은 같습니다.");  
    else  
        System.out.println("두 문자열은 다릅니다.");  
  
    cmp = st1.compareTo(st2);  
    if(cmp == 0)  
        System.out.println("두 문자열은 일치합니다.");  
    else if (cmp < 0)  
        System.out.println("사전의 앞에 위치하는 문자: " + st1);  
    else  
        System.out.println("사전의 앞에 위치하는 문자: " + st2);  
  
    if(st1.compareToIgnoreCase(st2) == 0)  
        System.out.println("두 문자열은 같습니다.");  
    else  
        System.out.println("두 문자열은 다릅니다.");  
}
```

명령 프롬프트

```
C:\JavaStudy>java CompString  
두 문자열은 다릅니다.  
사전의 앞에 위치하는 문자: Lexicographically  
두 문자열은 같습니다.  
  
C:\JavaStudy>
```

# 기본 자료형의 값을 문자열로 바꾸기

---

```
double e = 2.718281;
```

```
String se = String.valueOf(e);
```

```
static String valueOf(boolean b)
```

```
static String valueOf(char c)
```

```
static String valueOf(double d)
```

```
static String valueOf(float f)
```

```
static String valueOf(int i)
```

```
static String valueOf(long l)
```

# 문자열 대상 + 연산과 += 연산

---

```
System.out.println("funny" + "camp");
```



컴파일러에 의한 자동 변환

```
System.out.println("funny".concat("camp"));
```

```
String str = "funny";
```

```
str += "camp";    // str = str + "camp"
```



```
str = str.concat("camp")
```

# 문자열과 기본 자료형의 + 연산

---

```
String str = "age: " + 17;
```

↓ NO!

```
String str = "age: ".concat(17);
```

```
String str = "age: " + 17;
```

↓ YES!

```
String str = "age: ".concat(String.valueOf(17));
```



## concat 메소드는 이어서 호출 가능

---

```
String str = "AB".concat("CD").concat("EF");
```

```
→ String str = ("AB".concat("CD")).concat("EF");
```

```
→ String str = "ABCD".concat("EF");
```

```
→ String str = "ABCDEF";
```

# 문자열 결합의 최적화를 하지 않을 경우

---

```
String birth = "<양>" + 7 + '.' + 16;
```

너무 과도한 String 인스턴스 생성으로 이어진다.  
따라서 컴파일러는 이렇게 변환하지 않는다.

String birth =

```
"<양>".concat(String.valueOf(7)).concat(String.valueOf('.')).concat(String.valueOf(16));
```

이 문장에서 중간에 새로 생성되는 String 인스턴스의 수는? 많다~

# 문자열 결합의 최적화를 진행 할 경우

---

```
String birth = "<양>" + 7 + '.' + 16;
```



최종 결과물에 대한 인스턴스 생성 이외에 중간에 인스턴스 생성하지 않는다.  
따라서 컴파일러는 이 방식으로 변환을 진행한다.

```
String birth = (new StringBuilder("<양>").append(7).append('.').append(16)).toString();
```

이 문장에서 중간에 새로 생성되는 String 인스턴스의 수는? 딱 한 개!

`StringBuilder` append(String str)

`StringBuilder` append(double d)

`StringBuilder` append(int i)

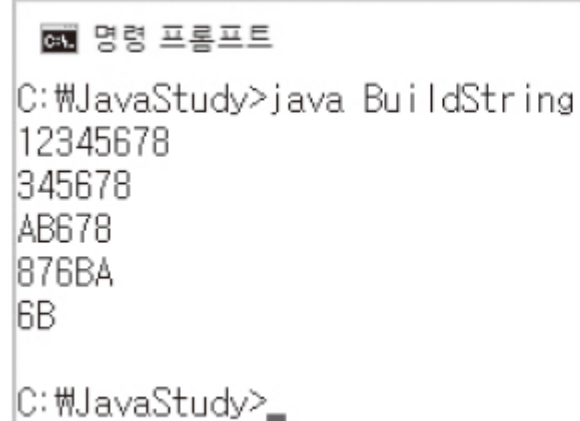
`StringBuilder` append(char c)

. . . 등등 다양하게 오버로딩 그리고 반환하는 값은 호출된 메소드가 속한 인스턴스의 참조 값

# StringBuilder

---

```
public static void main(String[] args) {  
    // 문자열 "123"이 저장된 인스턴스의 생성  
    StringBuilder stbuf = new StringBuilder("123");  
  
    stbuf.append(45678);    // 문자열 덧붙이기  
    System.out.println(stbuf.toString());  
  
    stbuf.delete(0, 2);    // 문자열 일부 삭제  
    System.out.println(stbuf.toString());  
  
    stbuf.replace(0, 3, "AB");    // 문자열 일부 교체  
    System.out.println(stbuf.toString());  
  
    stbuf.reverse();    // 문자열 내용 뒤집기  
    System.out.println(stbuf.toString());  
  
    String sub = stbuf.substring(2, 4);    // 일부만 문자열로 반환  
    System.out.println(sub);  
}
```



```
C:\JavaStudy>java BuildString  
12345678  
345678  
AB678  
876BA  
6B  
  
C:\JavaStudy>
```

# StringBuffer

---

StringBuffer와 StringBuilder는 기능적으로는 완전히 동일하다. 즉 다음 세 가지가 일치한다.

- 생성자를 포함한 메소드의 수
- 메소드의 기능
- 메소드의 이름과 매개변수의 선언

BUT!

- StringBuffer는 스레드에 안전하다.
- 따라서 스레드 안전성이 불필요한 상황에서 StringBuffer를 사용하면 성능의 저하만 유발하게 된다.
- 그래서 StringBuilder가 등장하게 되었다.