

# 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

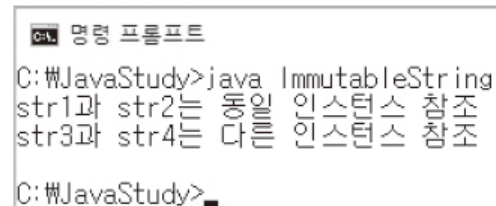
```
public static void main(String[] args) {  
    String str1 = new String("Simple String");  
    String str2 = "The Best String";  
  
    System.out.println(str1);  
    System.out.println(str1.length());  
    System.out.println();    // '개 행'  
  
    System.out.println(str2);  
    System.out.println(str2.length());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

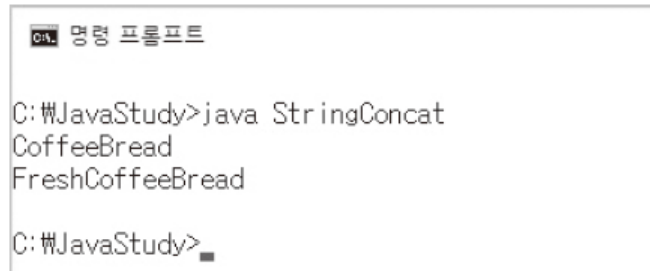
# String 인스턴스 기반 switch문 구성

```
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);  
    }  
}
```



CA. 명령 프롬프트

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



# 문자열의 일부 추출

```
String str = "abcdefg";
```

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

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

인덱스 2 이후의 내용으로 이뤄진 문자열 "cdefg" 반환

```
String str = "abcdefg";
```

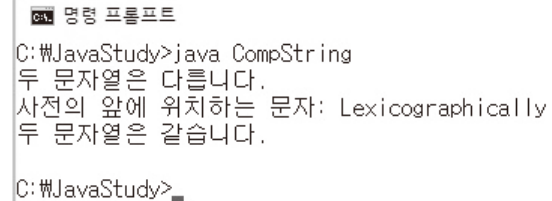
```
str.substring(2, 4);
```

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.");  
}
```



```
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 진행할 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);  
}
```

명령 프롬프트

```
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.