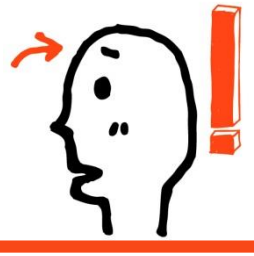
Android Java

10장 메소드 오버로딩과 String 클래스

매개변수 형(type)이 다르거나 개수가 다르거나



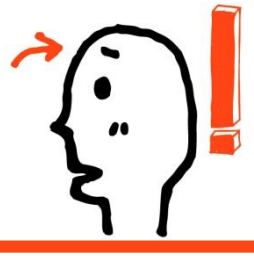
- 메소드 오버로딩이란 동일한 이름의 메소드를 둘 이상 동시에 정의하는 것을 뜻한다.
- 메소드의 매개변수 선언(개수 또는 자료형)이 다르면 메소드 오버로딩 성립
- 오버로딩 된 메소드는 호출 시 전달하는 인자를 통해서 구분된다.

```
class AAA                                  오버로딩 된 메소드
{
    void isYourFunc(int n) { . . . }
    void isYourFunc(int n1, int n2) { . . . }
    void isYourFunc(int n1, double n2) { . . . }
    . . . .
}
```

```
AAA inst = new AAA();
inst.isYourFunc(10);
inst.isYourFunc(10, 20);
inst.isYourFunc(12, 3.15);
```

전달되는 인자의 유형을 통해서 호출되는 함수가 결정!

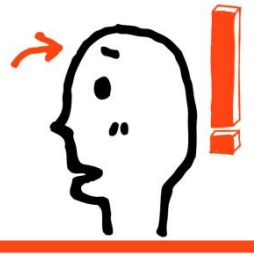
다형성을 구현하는 오버로딩



다형성을 구현하는 오버로딩

- **오버로딩** : 하나의 클래스에 같은 이름의 메소드들을 여러 가지 형태로 정의하는 것
- 하나의 클래스 안에서는 여러 메소드들을 구별하기 위하여 메소드 이름은 달라야 하지만 오버로딩으로 구현된 메소드들은 이 법칙에서 제외
- 오버로딩은 하나의 메소드 이름에 다양하게**(다형성)** 쌓아올리는 형태이기 때문
- 메소드 이름은 하나지만 여러 가지 형태를 갖음
- 생성자도 메소드의 한 종류이므로 오버로딩이 가능
- 단, 매개변수가 같지만 반환 데이터형이 다른 경우를 오버로딩이라고 착각하지 않도록 주의

다형성을 구현하는 오버로딩



다형성을 구현하는 오버로딩

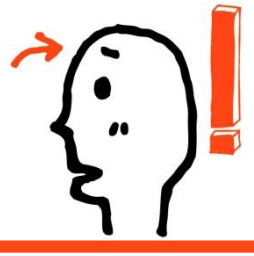
오버로딩의 사용 규칙

사용법 : 메소드 선언부의 반환형과 메소드 이름은 반드시 같아야 한다.
매개변수의 개수나 데이터형이 반드시 달라야 한다.

사용예 :
```
public void setStatus(int i) { ……};
public void setStatus(long l) { …… };
public void setStatus(Double d) { …… };
public void setStatus(String str) { …… };
public void setStatus(int i, String msg) { …… };
```

- setStatus( )라는 메소드는 이름은 모두 같으나 매개변수의 데이터형이 모두 다름

다형성을 구현하는 오버로딩



▶ 다형성을 구현하는 오버로딩

▷ 오버로딩을 사용함으로써 얻을 수 있는 장점

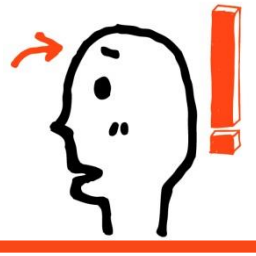
① 오버로딩된 메소드는 편리

→ 오버로딩된 메소드들이 있다면 형변환과 같은 부가적인 작업 없이 편리하게 개발할 수 있음

② 클래스에서 메소드 이름을 절약할 수 있음

→ 오버로딩이 없다면 같은 기능을 하더라도 메소드 이름을 중복해서 선언할 수 없으므로 메소드 이름이 복잡해질 수 밖에 없음

요런! 아주 기막히게 애매한 상황!



형변환의 규칙까지 적용해야만 메소드가 구분되는 기막히게 애매한 상황은 만들지 말자!

```
class AAA
{
    void isYourFunc(int n) { . . . }
    void isYourFunc(int n1, int n2) { . . . }
    void isYourFunc(int n1, double n2) { . . . }
     . . . .
}
```

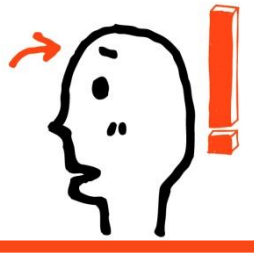
오버로딩 된 메소드

```
AAA inst = new AAA();
inst.isYourFunc(10, 'a');
```

무엇이 호출되겠는가? 문자 'a'는 int형으로도, double형으로도 변환이 가능하다!

결론적으로, 형변환 규칙을 적용하되 가장 가까운 위치의 자료형으로 변환이 이뤄진다. 따라서 is...(int n1, int n2)가 호출된다.

생성자도 오버로딩의 대상이 됩니다.



생성자의 오버로딩은 하나의 클래스를 기반으로 다양한 형태의 인스턴스 생성을 가능하게 한다.

```
class Person
{
    private int perID;
    private int milID;
    public Person(int pID, int mID)
    {
        perID=pID;
        milID=mID;
    }
    public Person(int pID)
    {
        perID=pID;
        milID=0;
    }
    public void showInfo()
    {
        System.out.println("민번 : "+perID);
        if(milID!=0)
            System.out.println("군번 : "+milID+'\n');
        else
            System.out.println("군과 관계 없음 \n");
    }
}
```

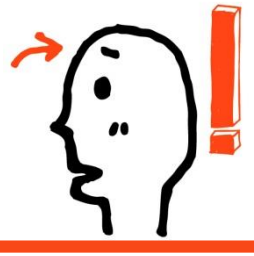
군 필자를 위한 생성자

군 미필자를 위한 생성자

```
public static void main(String[] args)
{
    Person man=new Person(950123, 880102);
    Person woman=new Person(941125);
    man.showInfo();
    woman.showInfo();
}
```

군을 제대한 남성과 여성을 의미하는 인스턴스의 생성이 가능하다!

메소드 오버로딩 성공과 실패 사례

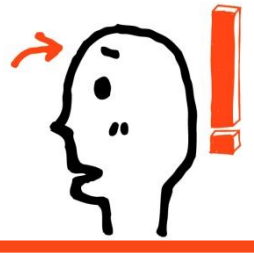


▷ 리턴 타입은 오버로딩과 관련 없음

```
// 메소드 오버로딩이 성공한 사례

class MethodOverloading {
    public int getSum(int i, int j) {
        return i + j;
    }
    public int getSum(int i, int j, int k) {
        return i + j + k;
    }
    public double getSum(double i, double j) {
        return i + j;
    }
}
```

```
// 메소드 오버로딩이 실패한 사례

class MethodOverloadingFail {
    public int getSum(int i, int j) {
        return i + j;
    }
    public double getSum(int i, int j) {
        return (double)(i + j);
    }
}
```

키워드 this를 이용한 다른 생성자의 호출



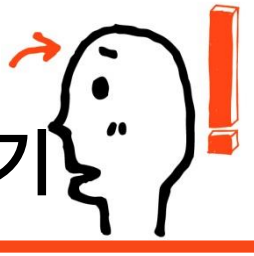
- 키워드 this를 이용하면 생성자 내에서 다른 생성자를 호출할 수 있다.
- 이는 생성자의 추가 정의에 대한 편의를 제공한다.
- 생성자마다 중복되는 초기화 과정의 중복을 피할 수 있다.

```
class Person
{
    private int perID;
    private int milID;
    private int birthYear;
    private int birthMonth;
    private int birthDay;

    public Person(int perID, int milID, int bYear, int bMonth, int bDay)
    {
        this.perID=perID;
        this.milID=milID;
        birthYear=bYear;
        birthMonth=bMonth;
        birthDay=bDay;
    }
    public Person(int pID, int bYear, int bMonth, int bDay)
    {
        this(pID, 0, bYear, bMonth, bDay);
    }
```

생성자의 재호출을 위한 키워드 this가 존재하지 않았다고 생각해 보자. 이 클래스의 생성자 정의에 어떠한 변화가 있어야 하는가?

인자로 pID, 0, bYear, bMonth, bDay를 전달받는 생성자의 호출문장

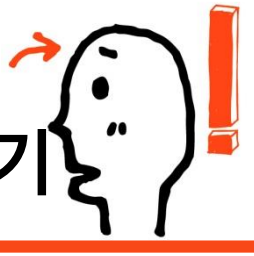
Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



코드 6-8 package com.gilbut.chapter6;

```
1   public class CoffeeValue
2   {
3       protected int capacity;
4       protected String coffeeName;
5
6       public CoffeeValue(String name, int size)
7       {
8           capacity = size;
9           coffeeName = name;
10      }
11
12      public String getInfo()
13      {
14          return "Capacity : " + capacity + "ml , " + "CoffeeName : " + coffeeName;
15      }
```

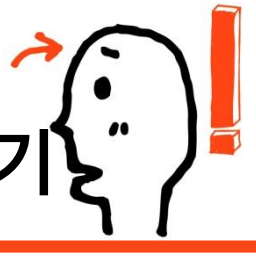
Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



```
16
17      public static void main(String[] args)
18      {
19          CoffeeValue order1 = new CoffeeValue("Americano", 360);
20          CoffeeValue order2 = new CoffeeValue("Cafe Latte", 500);
21          CoffeeValue order3 = new CoffeeValue("Cafe Mocha", 200);
22
23          System.out.println(System.identityHashCode(order1) + ", " + order1.
            getInfo());
24          System.out.println(System.identityHashCode(order2) + ", " + order2.
            getInfo());
25          System.out.println(System.identityHashCode(order3) + ", " + order3.
            getInfo());
26      }
27  }
```

이 3개의 객체는 모두 다른 JVM 메모리 위치에 저장되어 있기 때문에 클래스형은 같지만 실제 데이터는 다릅니다.

Super와 this 키워드를 사용해 명시적으로 객체 지칭하기

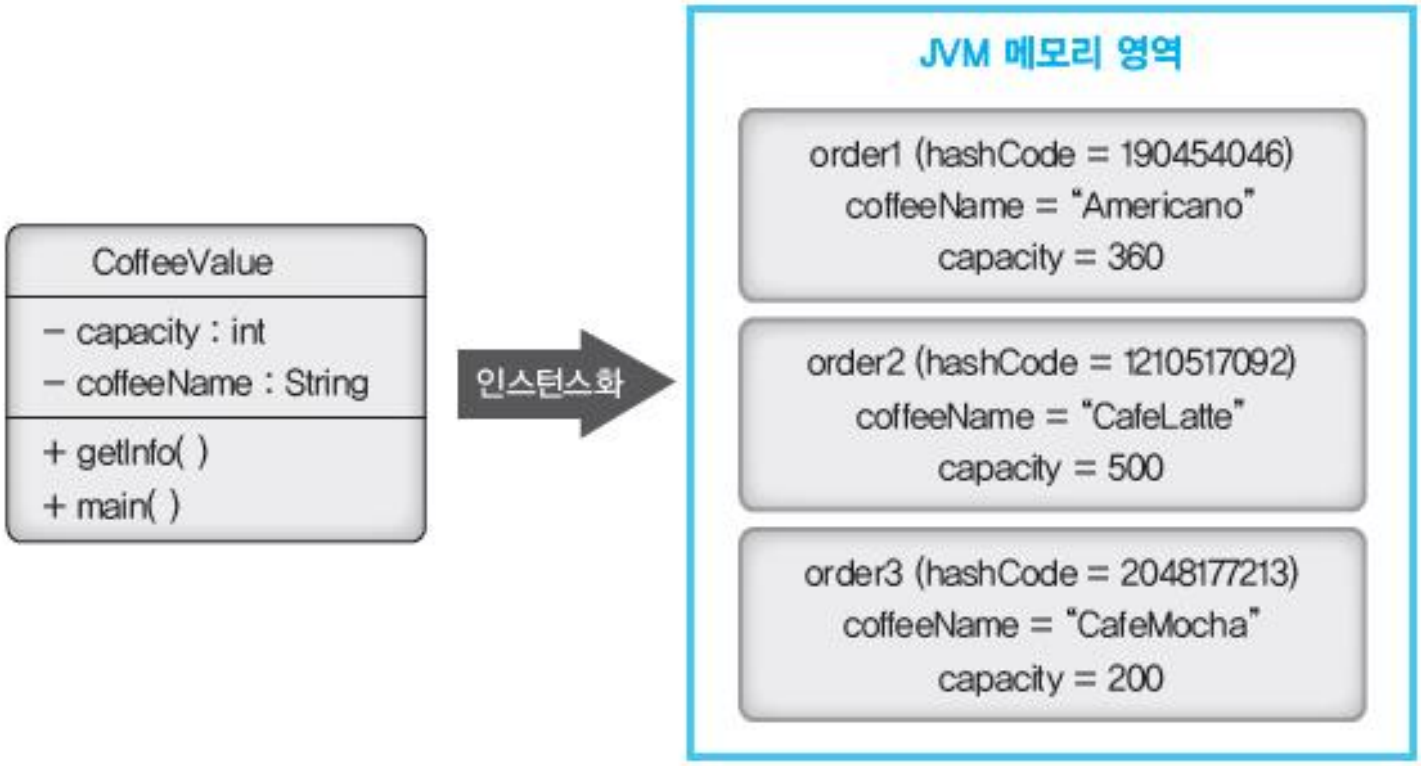


실행결과

```
190454046, Capacity : 360ml , CoffeeName : Americano
1210517092, Capacity : 500ml , CoffeeName : Cafe Latte
2048177213, Capacity : 200ml , CoffeeName : Cafe Mocha
```

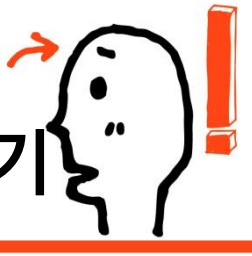
- 자바 클래스를 new 키워드를 사용해서 인스턴스화 할 때 JVM에서는 새로운 객체가 생성
- 객체는 JVM 메모리에 생성되며, 여러 번 인스턴스하면 각기 다른 객체들이 JVM 메모리에 생성
- order1, order2, order3 3개의 객체 모두 해시 코드가 다르기 때문에
  다른 메모리 공간에 저장되어 있으며 속성값 또한 다름

Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



△ 그림 6-9 CoffeeValue 클래스의 main 메소드를 실행한 결과

Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



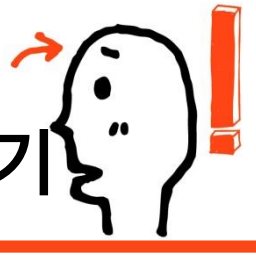
```
public CoffeeValue(String coffeeName, int capacity)
{
    capacity = capacity;
    coffeeName = coffeeName;
}
```

- 클래스 속성인 coffeeName 변수는 생성자의 실행부 밖에 선언되어 있으므로
  자바 컴파일러는 coffeeName = coffeeName 구문을
  매개변수에 다시 매개변수의 값을 대입하는 것으로 인식
  그러므로 클래스 변수에는 아무런 값도 대입되지 않음

실행결과

```
190454046, Capacity : 0ml , CoffeeName : null
1210517092, Capacity : 0ml , CoffeeName : null
2048177213, Capacity : 0ml , CoffeeName : null
```

Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



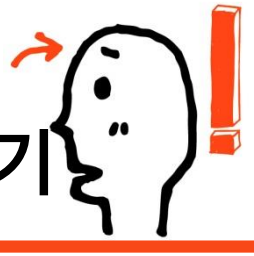
```
public CoffeeValue(String coffeeName, int capacity)
{
    this.capacity = capacity;
    this.coffeeName = coffeeName;
}
```

▷ 명시적으로 변수 이름이 어떤 변수를 의미하는지 구별하기 위해서 this 키워드를 활용

실행결과

```
190454046, Capacity : 360ml , CoffeeName : Americano
1210517092, Capacity : 500ml , CoffeeName : Cafe Latte
2048177213, Capacity : 200ml , CoffeeName : Cafe Mocha
```

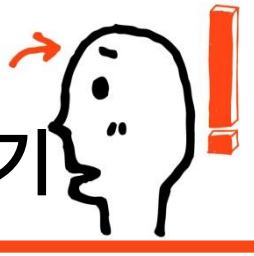
Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



super와 this 키워드 정의

- **this** 키워드 : 인스턴스화된 자기 자신의 객체를 지칭, 'this = 자신이 속한 객체'
- **super** 키워드 : 인스턴스화된 부모 객체를 지칭, 'super = 자신이 속한 객체의 부모 객체'
- this와 super 키워드는 클래스의 메소드나 클래스 변수와 결합해서 사용
- this와 super 키워드는 **객체를 대신해서** 쓰는 키워드
- JVM 메모리에는 같은 클래스형을 갖는 수많은 객체가 존재할 수 있는데,
  그 많은 객체 중에서 자기 자신 객체의 메소드나 클래스 변수를 가리키기 위해서 this 사용

Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



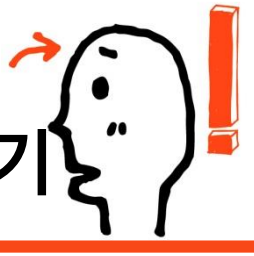
super와 this 키워드 정의

this와 super 접근자를 사용하는 방법

```
사용법 : this( );                    // 자신 객체의 생성자를 호출
         this.[변수 이름];            // 자신 객체의 변수를 호출
         this.[메소드 이름]( );       // 자신 객체의 메소드를 호출
사용예 : this();
         this.age;                   // 자신 객체의 age라는 변수를 참조한다.
         this.setAge();              // 자신 객체의 setAge() 메소드를 호출한다.
```

```
사용법 : super( );                   // 부모 객체의 생성자를 호출
         super.[변수 이름];           // 부모 객체의 변수를 호출
         super.[메소드 이름] ( );     // 부모 객체의 메소드를 호출
사용예 : super();
         super.age;                  // 부모 객체의 age라는 변수를 참조한다.
         super.setAge();             // 부모 객체의 setAge() 메소드를 호출한다.
```

Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



super와 this 키워드 정의

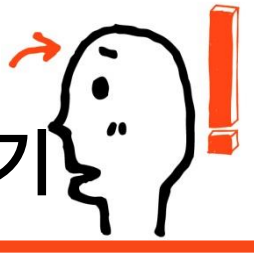
코드 6-9 package com.gilbut.chapter6;

```
1   public class CoffeeLatteValue extends CoffeeValue
2   {
3       protected int capacityMilk;
4
5       public CoffeeLatteValue(String coffeeName, int capacityTotal, int capacityMilk)
6       {
7           super(coffeeName, capacityTotal);
8           this.capacityMilk = capacityMilk;
9       }
10
11      public String getInfo()
12      {
13          return "Milk Capacity : " + (this.capacityMilk) + "ml";
14      }
15
```

부모 클래스의 생성자를 호출합니다.

자신의 capacityMilk 변수에 매개변수로 받은 capacityMilk 값을 대입합니다.

Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



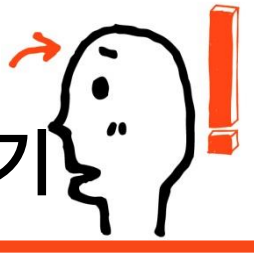
super와 this 키워드 정의

```
16      public String getDescription()
17      {
18          String rt = super.getInfo();
19          rt += "\n";
20          rt += this.getInfo();
21
22          return rt;
23      }
24
26      public static void main(String[] args)
27      {
28          CoffeeLatteValue order1 = new CoffeeLatteValue("Cafe Latte", 500, 100);
29          System.out.println(order1.getDescription());
30      }
31  }
```

부모 클래스의 getInfo( ) 메소드를 호출합니다.

오버라이드한 getInfo( ) 메소드를 호출합니다.

Super와 this 키워드를 사용해 명시적으로 객체 지칭하기



super와 this 키워드 정의

실행결과

```
Capacity : 500ml , CoffeeName : Cafe Latte
Milk Capacity : 100ml
```

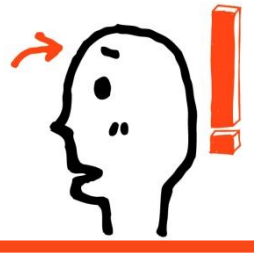
- 실행 결과 첫 번째 라인은 super.getInfo( ) 구문에 의해
  부모 클래스인 CoffeeValue 객체의 getInfo( ) 메소드를 호출했음을 알 수 있음

- 두 번째 라인은 자신(CoffeeLatteValue 클래스)의 getInfo( ) 메소드를 호출한 것

Android Java

10-1장 String 클래스

String 클래스의 인스턴스 생성



- JAVA는 큰 따옴표로 묶여서 표현되는 문자열을 모두 인스턴스화 한다.
- 문자열은 String 이라는 이름의 클래스로 표현된다.

```
String str1 = "String Instance";
String str2 = "My String";
```

두 개의 String 인스턴스 생성,
그리고 참조변수 str1과 str2로 참조

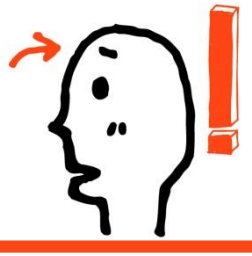
```
System.out.println("Hello JAVA!");
System.out.println("My Coffee");
```

println 메소드의 매개변수형이 String이
기 때문에 이러한 문장의 구성이 가능하다.

```
class StringInstance
{
    public static void main(String[] args)
    {
        java.lang.String str="My name is Sunny";

        int strLen1=str.length();
        System.out.println("길이 1 : "+strLen1);

        int strLen2="한글의 길이는 어떻게?".length();
        System.out.println("길이 2 : "+strLen2);
    }
}
```

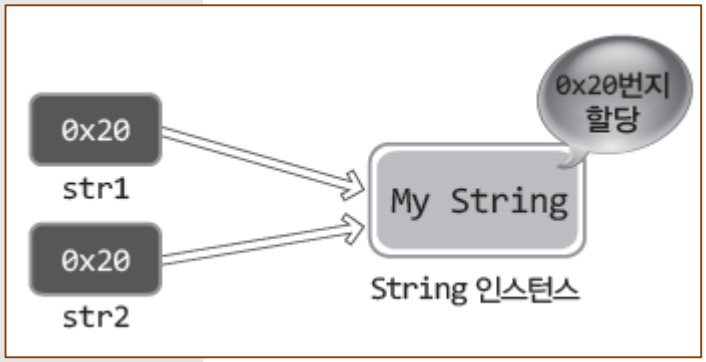
문자열의 선언은 인스턴스의 생성으로
이어짐을 보이는 문장

String 인스턴스는 상수 형태의 인스턴스이다.



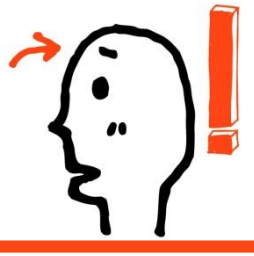
- String 인스턴스에 저장된 문자열의 내용은 변경이 불가능하다.
- 이는 동일한 문자열의 인스턴스를 하나만 생성해서 공유하기 위함이다.

```
public static void main(String[] args)
{
    String str1="My String";
    String str2="My String";
    String str3="Your String";

    if(str1==str2)
        System.out.println("동일 인스턴스 참조");
    else
        System.out.println("다른 인스턴스 참조");
```



String 인스턴스의 문자열 변경이 불가능하기 때문에 둘 이상의 참조변수가 동시에 참조를 해도 문제가 발생하지 않는다!

String을 사용한 기본 문자열 연산 익히기



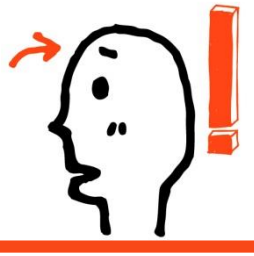
문자열 연산을 위한 기본 메소드

메소드	설명
concat(String str)	대상 문자열 뒤에 매개변수 str 문자열을 덧붙인 새로운 문자열을 반환한다. "ABC".concat("DEF"); ⇒ 결과 : "ABCDEF"
substring(int beginIndex)	대상 문자열에서 매개변수 beginIndex 위치에 있는 문자열을 새롭게 반환한다. 문자열을 자르는 기능을 한다. beginIndex는 0부터 시작한다. 예를 들어, 아래의 문자열 "ABCD" 중 A의 인덱스는 0, B의 인덱스는 1, C의 인덱스는 2, D의 인덱스는 3이다. 그러므로 "ABCD".substring(3)의 출력 결과는 D가 된다. "ABCD".substring(0); ⇒ 결과 : "ABCD" "ABCD".substring(3); ⇒ 결과 : "D"
substring(int beginIndex, int endIndex)	문자열에서 beginIndex와 endIndex 위치 사이에 있는 문자열을 새롭게 반환한다. 그러므로 endIndex의 값은 반드시 beginIndex의 값보다 커야 한다. 문자를 자르는 기능을 한다. "ABCDE".substring(3,5); ⇒ 결과 : "DE"

△ 표 3-12 문자열을 다루기 위한 메소드들

API Document의 참조를 통한 String 클래스의 인스턴스 메소드 관찰

String을 사용한 기본 문자열 연산 익히기



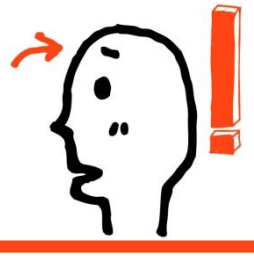
문자열 연산을 위한 기본 메소드

replace(Char oldChar, Char newChar)	대상 문자열에서 oldChar 문자를 newChar 문자로 바꾼 새로운 문자열을 반환한다. "ABCDE".replace('A', 'a'); ⇒ 결과 : "aBCDE"
replace(CharSequence t, CharSequence r)	대상 문자열에서 바꾸고 싶은 문자열 t를 문자열 r로 바꾸어 새로운 문자열을 반환한다. "ABCDE".replace("ABC", "abc"); ⇒ 결과 : "abcDE
toLowerCase( )	문자열의 문자들을 모두 소문자로 바꾸어 새로운 문자열을 반환한다. "Target String".toLowerCase( ); ⇒ 결과 : "target string"
toUpperCase( )	문자열의 문자들을 모두 대문자로 바꾸어 새로운 문자열을 반환한다. "Target String".toUpperCase( ); ⇒ 결과 : "TARGET STRING";

△ 표 3-12 문자열을 다루기 위한 메소드들

API Document의 참조를 통한 String 클래스의 인스턴스 메소드 관찰

String을 사용한 기본 문자열 연산 익히기



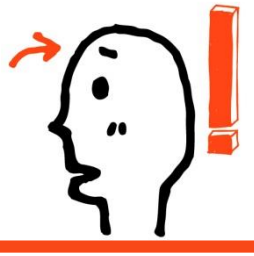
문자열 연산을 위한 기본 메소드

- 생성된 객체 데이터 즉, 문자열 바로 뒤 에 **'[메소드명](매개변수)'**와 같이 사용

```
String rt = "TEST".concat(" is very important");
System.out.println(rt); // 콘솔에 출력되는 결과는 TEST is very important이다.
```

- 객체 데이터는 TEST이며 is very important는 concat 메소드의 매개변수
- concat( ) 메소드는 객체 데이터와 매개변수를 덧붙여서 새로운 문자열을 반환하는 역할
- System.out.println(rt); 의 결과로 "TEST is very important" 문자열을 콘솔 화면에 출력

String을 사용한 기본 문자열 연산 익히기



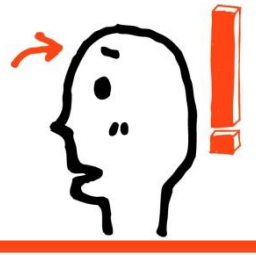
문자열 연산을 위한 기본 메소드

코드 3-9 package com.gilbut.chapter3;

```
1   public class StringOperation1
2   {
3      public static void main(String[] args)
4      {
5         String target = "Welcome to Java World";
6
7         System.out.println(target.concat(" and Gilbut press"));
8         System.out.println(target.substring(11));
9         System.out.println(target.substring(11, 16));
10        System.out.println(target.replace('o', 'O'));
11        System.out.println(target.replace("Java", "Gilbut"));
12        System.out.println(target.toLowerCase());
13        System.out.println(target.toUpperCase());
14     }
15  }
```

문자열 연산 메소드를 사용하기 위해서 반드시

String을 사용한 기본 문자열 연산 익히기

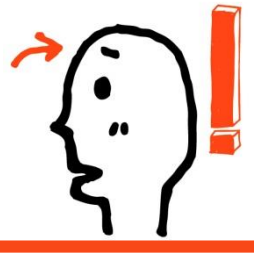


문자열 연산을 위한 기본 메소드

실행결과

```
Welcome to Java World and Gilbut press
Java World
Java
WelcOme tO Java WOrld
Welcome to Gilbut World
welcome to java world
WELCOME TO JAVA WORLD
```

String을 사용한 기본 문자열 연산 익히기



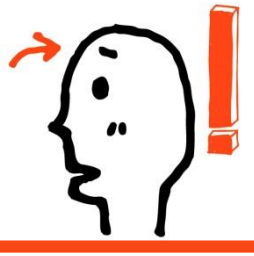
문자열을 비교하는 방법

- 문자열 데이터를 비교할 때는 기본형 데이터를 비교할 때처럼 **비교 연산자 '=='**를 사용
- **주의** : '==' 연산자가 실제로 비교하는 방법의 차이가 있음
- 개발자는 문자열의 철자가 같은지만 비교할 수도 있지만,
  JVM에 저장된 데이터의 위치가 같은지까지 비교할 수도 있음

"비교 연산자(==)는 변수의 데이터가 저장된 메모리의 위치를 서로 비교한다."

- 다시 말하면 '==' 연산자는 JVM 메모리에 저장된 데이터(객체)를 직접 비교하는 것

String을 사용한 기본 문자열 연산 익히기



문자열을 비교하는 방법

코드 3-10 package com.gilbut.chapter3;

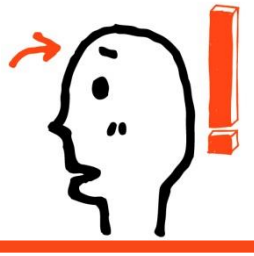
```
1   public class StringOperation2
2   {
3      public static void main(String[] args)
4      {
5
6         String coffee1 = "Americano";
7         String coffee2 = "Americano";
8         System.out.println("coffee1 and coffee2 : "
             + ((coffee1 == coffee2) ? "same" : "not same"));
9
10        String coffee3 = new String("Americano");
11        String coffee4 = new String("Americano");
12        System.out.println("coffee3 and coffee4 : "
             + ((coffee3 == coffee4) ? "same" : "not same"));
13
14        System.out.println(coffee1 + "," + System.identityHashCode(coffee1));
15        System.out.println(coffee2 + "," + System.identityHashCode(coffee2));
16        System.out.println(coffee3 + "," + System.identityHashCode(coffee3));
17        System.out.println(coffee4 + "," + System.identityHashCode(coffee4));
18     }
19  }
```

동일한 문자열 "Americano"를 참조하는 변수들

간단한 조건을 처리할 때 많이 사용되는 3항 연산자 구문

각각 새로운 문자열을 할당하는 변수들

String을 사용한 기본 문자열 연산 익히기



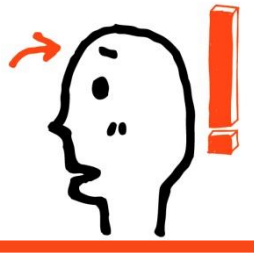
문자열을 비교하는 방법

실행결과

```
coffee1 and coffee2 : same
coffee3 and coffee4 : not same
Americano,1007247221
Americano,1007247221
Americano,190454046
Americano,1210517092
```

- coffee1과 coffee2 변수를 비교하는 구문에서는 서로 같음을 의미하는 결과(same)를 보여주고, coffee3와 coffee4 변수를 비교하는 구문에서는 서로 다름을 의미하는 결과(not same)를 보여줌
- 동일한 철자의 데이터라도 JVM에 생성된 데이터는 다를 수 있다는 뜻
- 문자열 철자를 비교할 때는 **equals( )**라는 메소드를 사용
  JVM 메모리에 위치한 문자열의 위치가 같은지 비교할 때는 **'==' 연산자**를 사용

String을 사용한 기본 문자열 연산 익히기

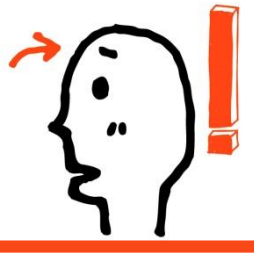


문자열을 비교하는 방법

메소드	설명
equals(Object anObject)	대상 문자열이 anObject 데이터(객체)와 같은지 비교해서 boolean 값을 반환한다. "ABC".equals("DEF"); ⇒ 결과 : false
equalsIgnoreCase(String anotherString)	대상 문자열이 anotherString과 대소문자 구별 없이 동일한지 확인한다. 반환값은 boolean 형이다. "teststring".equalsIgnoreCase("teststring"); ⇒ 결과 : true
compareTo(String anotherString)	대상 문자열이 사전적으로 앞에 있는지 뒤에 있는지 확인한다. 반환값이 0이면 두 문자열은 같고, 양수면 대상 문자열이 anotherString보다 뒤에 있고, 음수면 대상 문자열이 anotherString보다 앞에 있다. "a".compareTo("c"); ⇒ 결과 : -2
startWith(String prefix)	대상 문자열이 매개변수로 받은 접두사(prefix) 문자열로 시작하는지 확인한 후 boolean 값을 반환한다. "ABCDE".startWith("ABC"); ⇒ 결과 : true
endWith(String suffix)	대상 문자열이 매개변수로 받은 접미사(suffix) 문자열로 끝나는지 확인한 후 boolean 값을 반환한다. "ABCDE".endWith("DE"); ⇒ 결과 : true

△ 표 3-13 문자열을 비교하기 위한 메소드

String을 사용한 기본 문자열 연산 익히기



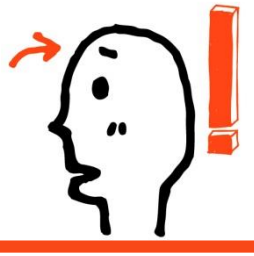
문자열을 비교하는 방법

코드 3-11 package com.gilbut.chapter3;

```
1   public class StringOperation3
2   {
3      public static void main(String[] args)
4      {
5         String coffee1 = new String("Americano");
6         String coffee2 = new String("Americano");
7         String coffee3 = new String("AmeRicAnO");
8         String coffee4 = new String("Blue mountin");
9         String coffee5 = new String("Cappuccino");
10
11        System.out.println((coffee1 == coffee2) ? "equal" : "not equal");
12        System.out.println((coffee1.equals(coffee2))  ? "equal" : "not equal");
13
14        System.out.println((coffee1.equals(coffee3)) ? "equal" : "not equal");
15        System.out.println((coffee1.equalsIgnoreCase(coffee3)) ? "equal" : "not
          equal");
16
17        System.out.println(coffee4.compareTo(coffee3));
18        System.out.println(coffee4.compareTo(coffee5));
19        System.out.println(coffee5.endsWith("A"));
20        System.out.println(coffee5.endsWith("no"));
21     }
22  }
```

철자가 같은 문자열을 비교할 때, '==' 연산자를 사용하면 대부분 false 값이 반환되어 당황합니다.
'==' 연산자는 JVM 메모리 위치까지 같은지 확인하고, equals( ) 메소드는 값만 같으면 true를 반환하는 것 기억하세요!

String을 사용한 기본 문자열 연산 익히기



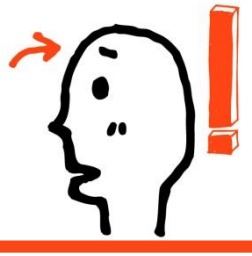
문자열을 비교하는 방법

실행결과

```
not equal
equal
not equal
equal
1
-1
false
true
```

- 첫 번째 라인은 **비교 연산자 '=='**을 사용하여 문자열을 비교한 결과로,
  coffee1과 coffee2 변수가 참조하고 있는 JVM 메모리 안의 데이터가 서로 같지 않음
- 두 번째 라인은 **equals( ) 메소드**를 사용해서 비교한 결과로,
  coffee1과 coffee2 변수의 문자열이 같음

String을 사용한 기본 문자열 연산 익히기



▶ 특정 문자열의 위치 파악

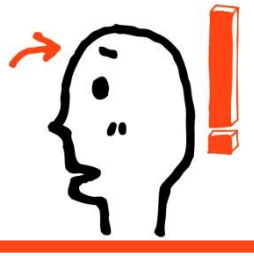
▷ 대상 문자열에서 특정 문자의 위치값을 나타내는 index를 알기 위한 메소드

→ **indexOf( )**와 **lastIndexOf( )**

메소드	설명
indexOf(String str)	대상 문자열에서 str 문자가 앞에서부터 맨 처음 시작하는 위치(index)를 반환한다. "ABCDE".indexOf("B"); ⇒ 결과 : 1
lastIndexOf(String str)	대상 문자열에서 str 문자가 뒤에서부터 맨 처음 시작하는 위치(index)를 반환한다. "ABCDEABC".lastIndexOf("A") ⇒ 결과 : 5

△ 표 3-14 대상 문자열에서 위치 파악을 위한 메소드

String을 사용한 기본 문자열 연산 익히기



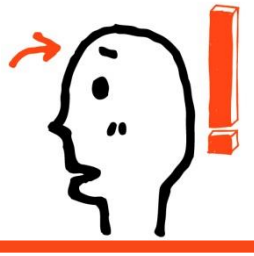
특정 문자열의 위치 파악

코드 3-12 package com.gilbut.chapter3;

```
1   public class StringOperation4
2   {
3      public static void main(String[] args)
4      {
5         if (args == null || args.length != 1)
6         {
7            System.out.println("Help : java StringOperation4 [Single character]");
8            return;
9         }
10
11        String alphabet = "abcdefghijklmnopqrstuvwxyz";
12        String number = "1234567890";
13
14        String str = args[0];
15        String temp = str.toLowerCase();
16
17        int alphabetIdx = alphabet.indexOf(temp);
18        int numberIdx = number.lastIndexOf(temp);
19
20        System.out.println("Input character : " + str);
21        System.out.println("Alphabet? : " + ((alphabetIdx >= 0) ? "true" : "false"));
22        System.out.println("Number? : " + ((numberIdx >= 0) ? "true" : "false"));
23
24     }
25  }
```

API Document의 참조를 통한 String 클래스의 인스턴스 메소드 관찰

String을 사용한 기본 문자열 연산 익히기



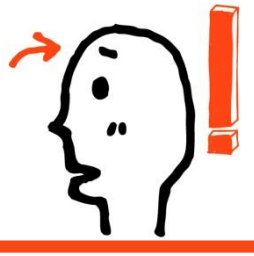
특정 문자열의 위치 파악

실행결과

```
Input character : 1
Alphabet? : false
Number? : true
```

- StringOperation4 예제는 입력 받은 문자가 숫자인지 알파벳 문자인지 확인하는 예제
- indexOf( ) 메소드의 결과값이 0 이상이면 temp 변수는 alphabet 문자 열 변수에 있음을 의미, 반대로 결과값이 음수면 temp 문자열은 알파벳이 아님을 의미

String을 사용한 기본 문자열 연산 익히기



문자열로 형 변환하는 법

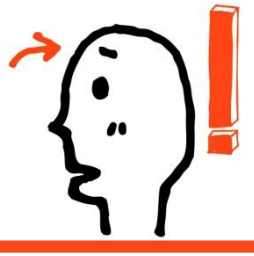
- 문자열 클래스인 String에서는 다양한 데이터들을 문자열로 바꿔주는 형 변환 함수인 **valueOf( ) 메소드**들을 제공
- valueOf( ) 메소드는 매개변수로 다양한 데이터들을 받을 수 있음

메소드	설명
valueOf(boolean b), valueOf(char C), valueOf(int i), valueOf(char c), valueOf(double d), valueOf(long l), valueOf(Object obj), valueOf(char[ ] data), valueOf(float f) 등	매개변수로 받는 데이터들을 문자열로 바꾸어 반환한다.

△ 표 3-15 문자열 형 변환을 위한 valueOf( ) 메소드

```
float f = 10.2031;
String temp = String.valueOf(f);
```

String을 사용한 기본 문자열 연산 익히기



▶ 특정 형식으로 문자열 포매팅하는 법

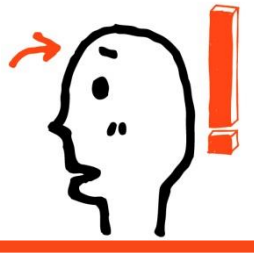
▷ **문자열 포매팅은?** 문자열의 형태를 정해진 문법에 의해서 변환하는 것

▷ 특정 형식으로 문자열을 출력하고 싶을 때 format( ) 메소드 사용

메소드	설명
format(String format, Object… args)	매개변수 format의 형태로 뒤따라 입력된 매개변수를 변경한다. 사용 방법은 String.format( );과 같이 문자열 뒤에 사용하면 된다.

△ 표 3-16 문자열을 지정한 형태로 변경시켜주는 format( ) 메소드

String을 사용한 기본 문자열 연산 익히기



특정 형식으로 문자열 포매팅하는 법

코드 3-13 package com.gilbut.chapter3;

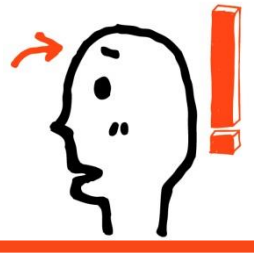
```
1   public class StringOperation5
2   {
3       public static void main(String[] args)
4       {
5           int i = 3;
6
7           System.out.println(i);
8           System.out.println(String.format("%04d", i));
9       }
10  }
```

실행결과

```
3
0003
```

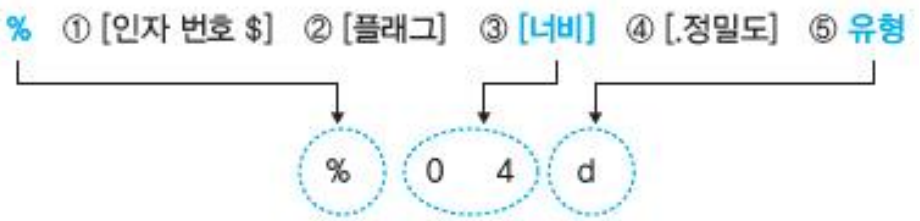
변수 i를 format( ) 메소드를 이용해서 포매팅한 결과를 화면에 출력하는 예제

String을 사용한 기본 문자열 연산 익히기



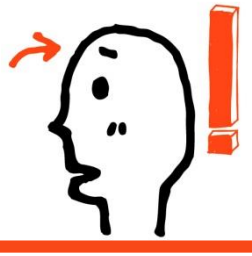
포맷 지시자

- format( ) 메소드에 사용한 %03d를 보통 **포맷 스트링** 혹은 **포맷 지시자**
- 대상 문자열 을 어떤 형식(포맷)으로 변환하도록 기호로 지정하는 역할



- 포맷 지시자는 여러 문자로 구성
- 개발자는 각각의 문자들을 조합해서 어떻게 포매팅할 것인지를 지정할 수 있음
- 포맷 지시자는 반드시 **'%'** 문자로 시작, 그 다음부터는 필요한 문자만 조합해서 사용

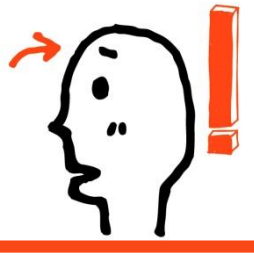
String을 사용한 기본 문자열 연산 익히기



포맷 지시자

- 포맷 지시자는 여러 대상 문자열 중 하나를 선택해서 포매팅 가능
  - 포맷 지시자를 구성하는 각각의 문자
    1. **인자 번호** format( ) 메소드에서 포매팅할 대상 문자열의 인자 번호를 의미
    2. **플래그** 숫자를 정렬하거나 특수한 옵션을 지정
    3. **너비** 포매팅할 대상의 너비를 지정. 너비는 결과값의 너비를 결정
    4. **정밀도** 소수점 아래에 대한 정밀도를 지정하며 대상이 float형인 경우에만 사용
    5. **유형** 포매팅할 유형을 의미

String을 사용한 기본 문자열 연산 익히기



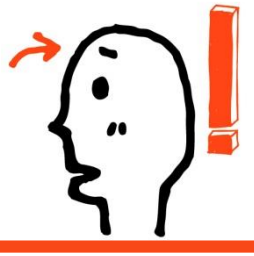
포맷 지시자

코드 3-14 package com.gilbut.chapter3;

```
1   public class StringOperation6
2   {
3       public static void main(String[] args)
4       {
5           System.out.println("돈과 관련된 유용한 format");
6           System.out.println(String.format("%,d", 1000000));
7           System.out.println(String.format("%,.2f", 1000000F));
8
9           System.out.println("\n문자열 자릿수 맞추는데 유용한 format");
10          System.out.println(String.format("%10s", "abcde"));
11          System.out.println(String.format("%10s", "abcdefghijklmnopqr"));
12
13          System.out.println("\n부동 소수점에 대한 유용한 format");
14          System.out.println(String.format("%.2f", 12345.121245));
15          System.out.println(String.format("%.2f", 12.1));
16      }
17  }
```

API Document의 참조를 통한 String 클래스의 인스턴스 메소드 관찰

String을 사용한 기본 문자열 연산 익히기

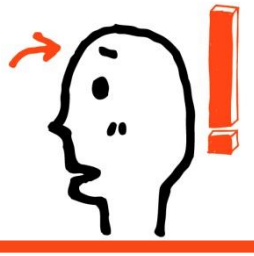


포맷 지시자

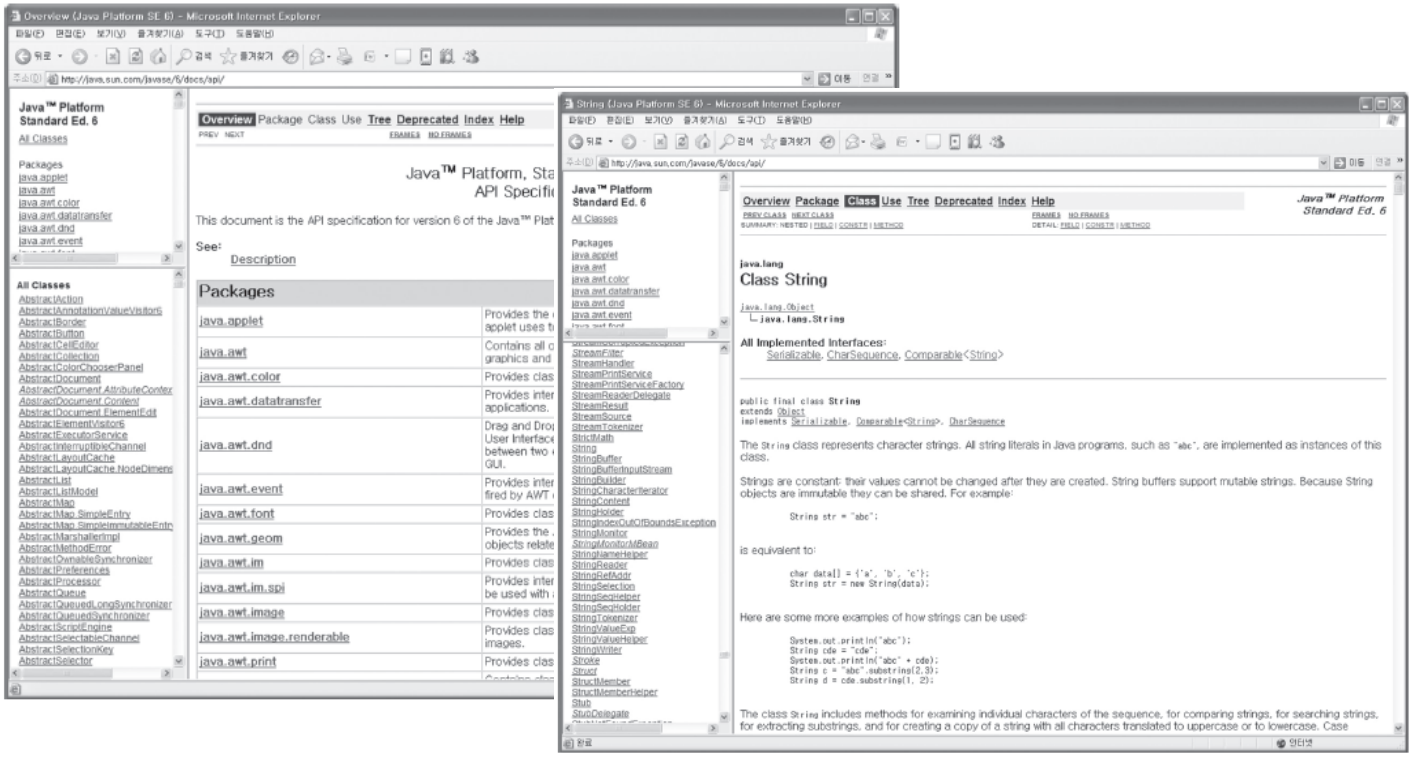
실행결과

```
돈과 관련된 유용한 format
1,000,000
1,000,000.00
문자열 자릿수 맞추는데 유용한 format
      abcde
abcdefghijklmnopqr

부동 소수점에 대한 유용한 format
12345.12
12.10
```

생성자를 뺀 메소드의 수만 50개가 넘습니다

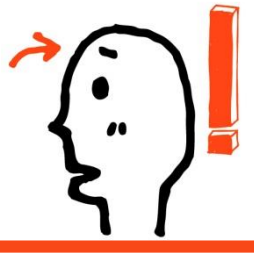


- 책에서 메소드의 기능을 찾는 습관은 조금씩 버려야 한다.
- API 문서를 볼 줄 모르는 자바 개발자는 있을 수 없다.
- API 문서를 참조하지 않고 개발하는 자바 개발자도 있을 수 없다.



API Document의 참조를 통한 String 클래스의 인스턴스 메소드 관찰

String 클래스가 제공하는 유용한 메소드들



- 문자열의 길이 반환 public int length()
- 두 문자열의 결합 public String concat(String str)
- 두 문자열의 비교 public int compareTo(String anotherString)

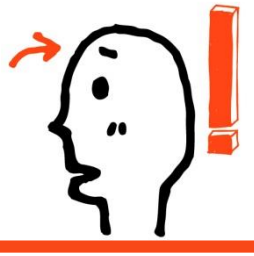
위 메소드의 사용방법만이라도 API 문서에서 확인해 보자!

```
class StringMethod
{
    public static void main(String[] args)
    {
        String str1="Smart";
        String str2=" and ";
        String str3="Simple";
        String str4=str1.concat(str2).concat(str3);

        System.out.println(str4);
        System.out.println("문자열 길이 : "+str4.length());

        if(str1.compareTo(str3)<0)
            System.out.println("str1이 앞선다");
        else
            System.out.println("str3이 앞선다");
    }
}
```

실행결과

```
Smart and Simple
문자열 길이 : 16
str3이 앞선다
```

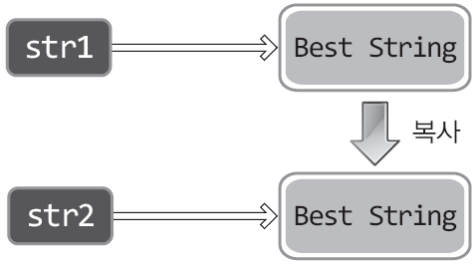
자바에서의 문자열 복사!



• 자바에서는 문자열을 상수의 형태로 관리하고, 또 동일한 유형의 문자열을 둘 이상 유지하지 않으므로 문자열의 복사라는 표현이 흔하지 않다.
• 무엇보다도 자바에서는 문자열을 복사가 필요 없다.

그러나 원하는 것이 인스턴스를 새로 생성해서 문자열의 내용을 그대로 복사하는 것이라면 다음과 같이 코드를 구성하면 된다.

```
String str1="Best String";
String str2=new String(str1);
```



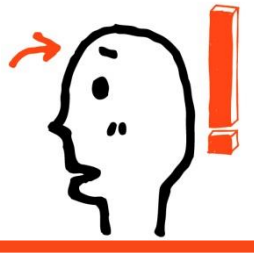

```
public String(String original)
```

새로운 문자열 인스턴스의 생성에 사용되는 생성자

```
class StringCopy
{
    public static void main(String[] args)
    {
        String str1="Lemon";
        String str2="Lemon";
        String str3=new String(str2);

        if(str1==str2)
            System.out.println("  실행  ");
        else
            System.out.println("  .  .  .  ");

        if(str2==str3)
            System.out.println("  .  .  .  ");
        else
            System.out.println("  실행  ");
    }
}
```

비교 연산자는 참조 값 비교!

+ 연산과 += 연산의 진실



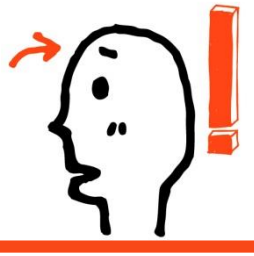
```
public static void main(String[] args)
{
    String str1="Lemon"+"ade";
    String str2="Lemon"+'A';
    String str3="Lemon"+3;
    String str4=1+"Lemon"+2;
    Str4+='!';

    System.out.println(str1);
    System.out.println(str2);
    System.out.println(str3);
    System.out.println(str4);
}
```

String str1="Lemon".concat("ade");
String str2="Lemon".concat(String.valueOf('A'));
String str3="Lemon".concat(String.valueOf(3));

위 예제의 str4의 선언이 다음과 같이 처리된다면?
String str4=Strimg.valueOf(1).concat("Lemon").concat(String.valueOf(2));

아무리 많은 + 연산을 취하더라도 딱 두 개의 인스턴스만 생성된다. StringBuilder 클래스의 도움으로...

StringBuilder



- StringBuilder는 문자열의 저장 및 변경을 위한 메모리 공간을 지니는 클래스
- 문자열 데이터의 추가를 위한 append와 삽입을 위한 insert 메소드 제공

```
class BuilderString
{
    public static void main(String[] args)
    {
        StringBuilder strBuf=new StringBuilder("AB");
        strBuf.append(25);
        strBuf.append('Y').append(true);
        System.out.println(strBuf);
        strBuf.insert(2, false);
        strBuf.insert(strBuf.length(), 'Z');
        System.out.println(strBuf);
    }
}
```

buf: AB

buf: AB25

buf: AB25Ytrue

buf: ABfalse25Ytrue

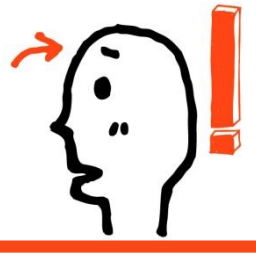
buf: ABfalse25YtrueZ

실행결과

```
AB25Ytrue
ABfalse25YtrueZ
```

연속해서 함수호출이 가능한 이유는
append 메소드가 strBuf의 참조 값을 반환하기 때문이다.

참조를 반환하는 메소드



- this의 반환은 인스턴스 자신의 참조 값 반환을 의미한다.
- 그리고 이렇게 반환되는 참조 값을 대상으로 연이은 함수호출이 가능하다.

```
class SimpleAdder
{
    private int num;
    public SimpleAdder() {num=0;}
    public SimpleAdder add(int num)
    {
        this.num+=num;
        return this;
    }
    public void showResult()
    {
        System.out.println("add result : "+num);
    }
}
```

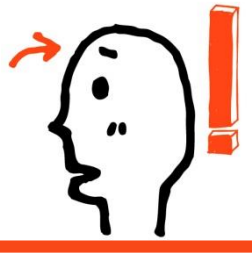
```
public static void main(String[] args)
{
    SimpleAdder adder=new SimpleAdder();
    adder.add(1).add(3).add(5).showResult();
}
```

실행결과

```
add result : 9
```

add 함수는 adder의 참조 값을 반환한다.

StringBuilder의 버퍼와 문자열 조합



- 추가되는 데이터 크기에 따라서 버퍼의 크기가 자동으로 확장된다.
- 생성자를 통해서 초기 버퍼의 크기를 지정할 수 있다.

- public StringBuilder() 기본 16개의 문자저장 버퍼 생성
- public StringBuilder(int capacity) capacity개의 문자저장 버퍼 생성
- public StringBuilder(String str) str.length()+16 크기의 버퍼 생성

문자열의 복잡한 조합의 과정에서는 StringBuilder의 인스턴스가 활용된다.
때문에 추가로 생성되는 인스턴스의 수는 최대 두 개이다!

```
String str4=1+"Lemon"+2;
```

```
new StringBuilder().append(1).append("Lemon").append(2).toString();
```

StringBuilder 인스턴스의 생성에서 한 개

toString 메소드의 호출에 의해서 한 개

THANK YOU

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술