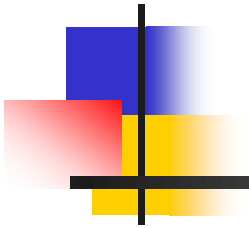
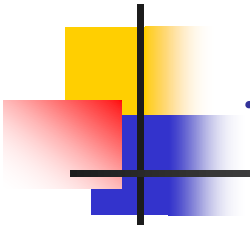


Java 15강 기타 문법, 내부클래스

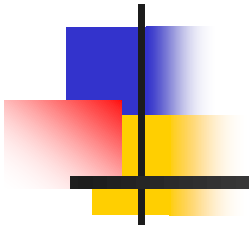
양 명 숙
[now4ever7@gmail.com]



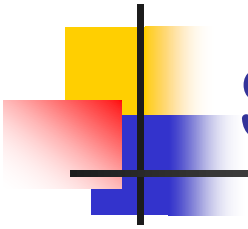
목차

- StringBuffer/StringBuilder
- StringTokenizer
- Object
- static import문

- 내부 클래스/익명클래스



StringBuffer/StringBuilder



StringBuffer

- String – 변경이 불가능한 문자열의 표현을 위한 클래스
- StringBuffer, StringBuilder – 변경이 가능한 문자열의 표현을 위한 클래스
- StringBuffer
  - 내부적으로 문자열 편집을 위한 버퍼(buffer)를 가지고 있으며, StringBuffer 인스턴스를 생성할 때 그 크기를 지정할 수 있음
  - 문자열의 저장 및 변경을 위한 메모리 공간(버퍼)을 내부에 지니는데, 이 메모리 공간은 그 크기가 자동으로 조절된다는 특징이 있음
  - append(), insert() 메서드가 가장 중요함

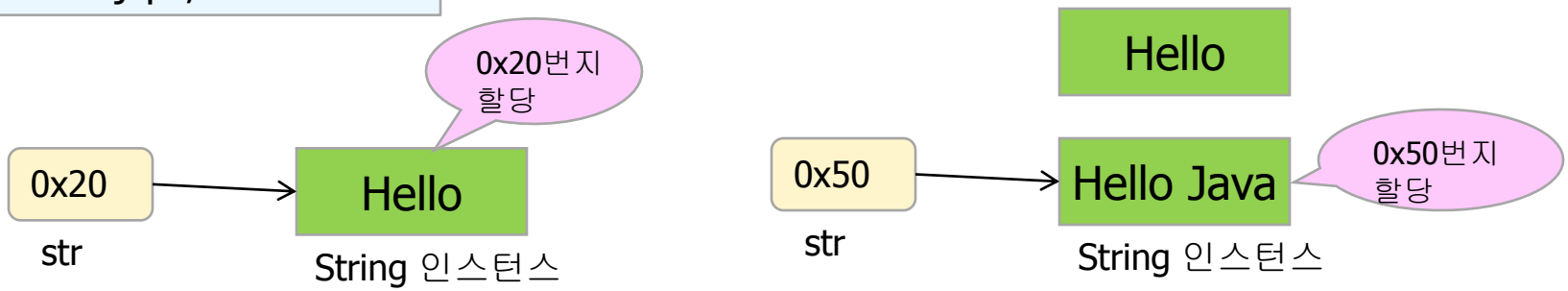
StringBuffer

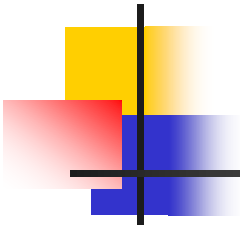
```
String  str = "Hello";
String  str = new String("Hello");
=> String 인스턴스가 생성됨
```

- String – 한번 만들어지면 더 이상 **그 값을 바꿀 수 없다**
  - String 객체는 변하지 않음
  - String 문자열을 더하면 새로운 String 객체가 생성되고, 기존 객체는 버려짐
  - 하나의 String을 만들어 더하는 작업을 한다면, 쓰레기를 만들게 됨

```
String  str = "Hello";
str = str + " java";
str = "jsp";
```

"Hello" 라는 단어를 갖고 있는 객체는 더 이상 사용할 수 없다
(쓰레기가 되며, 나중에 가비지 컬렉션의 대상이 됨)





StringBuffer

- StringBuffer, StringBuilder
  - String 클래스의 단점을 보완하기 위해서 나온 클래스

> StringBuffer – Thread safe 하다, 더 안전함
>
> StringBuilder - Thread safe 하지 않다, 속도는 더 빠름
> JDK 5.0 에 추가됨

  - 문자열을 더하더라도 새로운 객체를 생성하지 않음
  - + 기호를 사용하는 것이 아니라 append() 메서드 사용

StringBuilder

```
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" java");
```

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" java");
```

- append() 메서드를 여러 개 붙여서 사용 가능

```
StringBuilder sb = new StringBuilder();
sb.append("Hello").append(" java");
```

append() 메서드를 수행한 후에는 해당 StringBuilder 객체가 리턴되므로,
그 객체에 계속 붙이는 작업을 해도 무방함

```
public StringBuilder append(String str)
```

- JDK 5.0 이상에서는 String 의 더하기 연산을 할 경우, 컴파일할 때 자동으로 해당 연산을 StringBuilder 로 변환해줌
  - for 루프와 같이 반복 연산을 할 때에는 자동으로 변환 해주지 않음

예제

```
AB25Ytrue
ABfalse25YtrueZ
```

```
class BuilderTest{
    public static void main(String[] args){
        StringBuilder sb=new StringBuilder("AB");
        sb.append(25);
        sb.append('Y').append(true);
        System.out.println(sb);

        sb.insert(2, false);
        sb.insert(sb.length(), 'Z');
        System.out.println(sb);
    }
}
```

• append() 메서드
- 전달된 값을 StringBuilder 의 인스턴스가 저장하고 있는 문자열 데이터의 끝에 문자의 형태로 추가함

• insert(2, "값") 메서드
- 위치가 2인 지점에 , 두 번째 매개변수를 문자형태로 저장

• length() 메서드
- 저장된 문자의 개수 정보를 반환

public StringBuilder insert(int offset, String str)

1. String을 StringBuilder 로
String str="java";
StringBuilder sb = new StringBuilder(str);
System.out.println(sb);

2. StringBuilder를 String으로
StringBuilder sb = new StringBuilder("안녕");
String str = sb.toString();
System.out.println(str);

StringBuilder

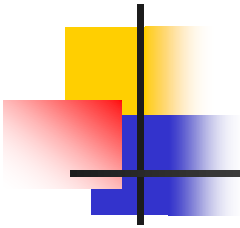
- StringBuilder 클래스는 String 클래스와 같이 문자열을 저장하기 위한 char형 배열의 참조 변수를 인스턴스 변수로 선언해 놓고 있다

```
public final class StringBuilder implements java.io.Serializable{
      private char[] value;
      ...
}
```

  - StringBuilder 클래스의 인스턴스를 생성할 때, 적절한 크기의 char형 배열이 생성되고, 이 배열은 문자열을 저장하고 편집하기 위한 공간(buffer)으로 사용됨

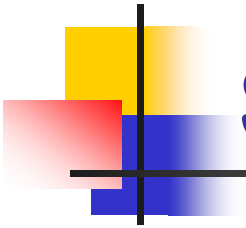
StringBuilder 인스턴스를 생성할 때는 생성자 StringBuilder(int length) 를 사용해서 StringBuilder 인스턴스에 저장될 문자열의 크기를 고려하여 **충분히 여유 있는 크기로 지정**하는 것이 좋다

편집 중인 문자열이 버퍼의 크기를 넘어서게 되면 버퍼의 크기를 늘려주는 작업이 추가로 수행되어야 하기 때문에 작업 효율이 떨어짐



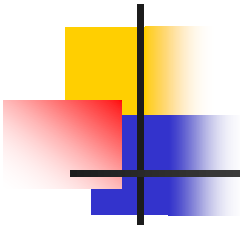
StringBuilder

- StringBuilder 의 내부에 존재하는 버퍼는 자동으로 크기가 증가하도록 설계되어 있음
  - 필요에 따라서는 그 크기를 조절할 수도 있음
  - 필요로 하는 버퍼의 크기를 미리 할당하는 것이 성능에 도움이 됨
- 생성자
  - StringBuilder() //16개의 문자 저장 버퍼 생성
    - 빈 버퍼 상태의 StringBuilder 인스턴스를 생성할 때 사용됨
    - 초기의 버퍼 크기 16, 문자가 저장됨에 따라서 자동으로 증가됨
  - StringBuilder(int capacity) //capacity개의 문자 저장 버퍼 생성
    - 초기 버퍼 크기를 지정할 때 사용
  - StringBuilder(String str) //str.length() + 16 개의 문자 저장 버퍼 생성
    - 문자열 정보를 저장하는 인스턴스의 생성에 사용



StringBuilder

- StringBuilder
  - JDK 5.0 에 추가됨
  - StringBuilder 는 StringBuffer 와 완전히 동일한 클래스
  - 단, 동기화(synchronization)처리를 하지 않기 때문에 멀티쓰레드 프로그래밍에서는 사용하면 안 되지만,
  - 멀티쓰레드 프로그래밍이 아닌 경우에는 StringBuffer 보다 빠른 성능을 보장함
  - 동기화의 여부를 제외하고는 두 클래스가 기능상으로 완전히 동일
- StringBuilder 사용하는 경우
  - 하나의 메소드 내에서 문자열을 생성하여 더할 경우에는 StringBuilder를 사용해도 됨
- StringBuffer 사용하는 경우
  - 멀티쓰레드 프로그래밍에서는 StringBuffer를 사용
  - 어떤 클래스에 문자열을 생성하여 더하기 위한 문자열을 처리하기 위한 인스턴스 변수가 선언되었고, 여러 쓰레드에서 이 변수를 동시에 접근하는 일이 있을 경우에는 반드시 StringBuffer를 사용해야 함



실습

- StringBuilder API 참조
- 1. 다음의 형태로 String 인스턴스를 하나 생성한 후 이 문자열을 역순으로 다시 출력하기
  - String str="ABCDEFG";
    - StringBuilder의 reverse() 메서드 이용
- 2. 다음의 형태로 주민번호를 담고 있는 String 인스턴스를 하나 생성한 후, 이 문자열을 활용하여 중간에 삽입된 '-'를 삭제한 String 인스턴스를 생성해보자
  - String str="990107-1112222";
    - StringBuilder의 lastIndexOf(), deleteCharAt() 메서드 이용
    - 또는 StringBuilder의 charAt(), deleteCharAt() 메서드 이용 – for문에서 사용

```
String str1="Hello";  //String str1 = new String("Hello");
String str2 = "Java";
=> String 인스턴스가 생성됨
```

String 클래스

- String 클래스의 인스턴스는 상수 형태의 인스턴스
  - String의 인스턴스는 상수의 성격을 갖는다
  - String의 인스턴스에 저장된 문자열의 데이터의 변경이 불가능하기 때문
- String
  - 저장된 문자열 데이터는 변경이 불가능함
  - 문자열을 표현할 때마다 인스턴스가 생성되니, 인스턴스의 생성을 최소화할 필요가 있었고,
  - 다음 원칙을 기준으로 인스턴스가 생성되도록 String 클래스를 정의하였음

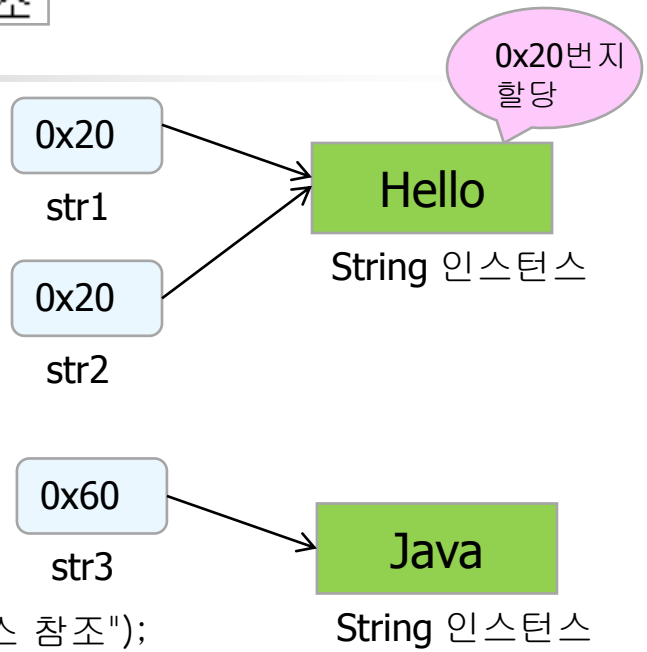
문자열이 동일한 경우에는 하나의 **String** 인스턴스만 생성해서 공유한다**!**

이를 통한 문제의 발생을 막기 위해서 String 인스턴스의 데이터 변경은 허용하지 않고 있다.

예제

```
str1, str2는 동일 인스턴스 참조
str2, str3는 다른 인스턴스 참조
str4, str5는 다른 인스턴스 참조
str1, str4는 다른 인스턴스 참조
```

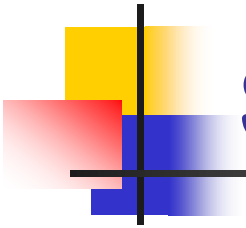
String 인스턴스의 공유




```
class ImmutableString
{
    public static void main(String[] args)
    {
        String str1="Hello";
        String str2="Hello";
        String str3="Java";

        if(str1==str2)  //참조자료형에서는 == 이 주소값 비교
            System.out.println("str1, str2는 동일 인스턴스 참조");
        else
            System.out.println("str1, str2는 다른 인스턴스 참조");

        if(str2==str3)
            System.out.println("str2, str3는 동일 인스턴스 참조");
        else
            System.out.println("str2, str3는 다른 인스턴스 참조");
```

예제

```
        String str4 = new String("Hello");
        String str5 = new String("Hello");

        if(str4==str5)
                System.out.println("str4, str5는 동일 인스턴스 참조");
        else
                System.out.println("str4, str5는 다른 인스턴스 참조");

        if(str1==str4)
                System.out.println("str1, str4는 동일 인스턴스 참조");
        else
                System.out.println("str1, str4는 다른 인스턴스 참조");

    }
}
```



String 클래스

- concat() 메서드

```
public String concat(String str)
```

  - 두 문자열을 결합함
  - 서로 다른 두 개의 문자열을 이어서 새로운 하나의 String 인스턴스가 생성됨
  - 이 메소드가 반환하는 값은 새롭게 생성된 String 인스턴스의 참조 값임

```
Happy and Smile
str1이 앞선다
```

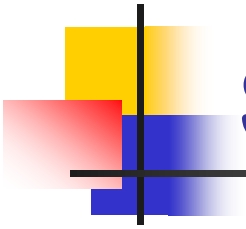
예제

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

        System.out.println(str4);

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

총 3개의 문자열을 하나로 묶어줌
이를 위해서 추가로 생성된 인스턴스의 수가 2개

public int compareTo(String anotherString)

=> this.charAt(k)-anotherString.charAt(k)



String의 + 연산

- 아무리 많은 + 연산을 하더라도, 추가적인 인스턴스의 생성은 두 개로 제한됨

```
String str4 = 1+ "Hello" + 2;
⇒String str4 = new StringBuilder().append(1).append("Hello").append(2).toString();
```

  - StringBuilder 인스턴스가 하나 생성
  - toString() 메서드는 StringBuilder 인스턴스가 저장하고 있는 문자 데이터들을 하나로 모아서 String 인스턴스를 생성

StringTokenizer

토큰(token)
- 일련의 문자열에서 구분할 수 있는 단위

- StringTokenizer
  - 긴 문자열을 지정된 구분자를 기준으로 토큰(token) 이라는 여러 개의 작은 문자열로 잘라내는 데 사용됨
    - 예) "10,20,30,40" 이라는 문자열이 있을 때 ',' 를 구분자로 잘라내면 "10", "20", "30", "40" 이라는 4개의 문자열(토큰)을 얻을 수 있다
  - String 클래스의 split() 메서드를 사용한 것과 유사

StringTokenizer 는 구분자로 단 **하나의 문자** 밖에 사용하지 못함

StringTokenizer

생성자 / 메서드	설 명
StringTokenizer(String str, String delim)	문자열(str)을 지정된 구분자(delim)로 나누는 StringTokenizer를 생성한다.(구분자는 토큰으로 간주되지 않음)
StringTokenizer(String str, String delim, boolean returnDelims)	문자열(str)을 지정된 구분자(delim)로 나누는 StringTokenizer를 생성한다. returnDelims의 값을 true로 하면 구분자도 토큰으로 간주된다
int countTokens()	전체 토큰의 수를 반환한다.
boolean hasMoreTokens()	토큰이 남아있는지 알려준다.
String nextToken()	다음 토큰을 반환한다.

예제 1

• StringTokenizer 는 단 한 문자의 구분자만 사용할 수 있기 때문에, "+-*/=()" 전체가 하나의 구분자가 아니라 각각의 문자가 모두 구분자임

• 두 문자 이상의 구분자를 사용해야 한다면 split() 메서드 이용

```
100
200
300
400
```

```
import java.util.*;

class StringTokenizerEx1 {
	public static void main(String[] args) {
		String source = "100,200,300,400";
		StringTokenizer st = new StringTokenizer(source, ",");

		while(st.hasMoreTokens()){
			System.out.println(st.nextToken());
		}

		String expression = "x=100*(200+300)/2";
		StringTokenizer st2 = new StringTokenizer(expression, "+-*/=()", true); //구분자도 토큰으로 간주

		while(st2.hasMoreTokens()){
			System.out.println(st2.nextToken());
		}
	}
}
```

```
x
=
100
*
(
200
+
300
)
/
2
```

예제2

• 문자열에 포함된 데이터가 두 가지 종류의 구분자로 나뉘어져 있을 때 두 개의 StringTokenizer 와 이중 반복문을 사용해서 처리하는 예제

• 한 학생의 정보를 구분하기 위해 "|" 를 사용
• 학생의 이름과 점수 등을 구분하기 위해 "," 를 사용

```
1
김천재
100
100
100
------
2
박수재
95
80
90
------
3
이자바
80
90
90
------
```

```
import java.util.*;

class StringTokenizerEx3 {
    public static void main(String args[]) {
        String source =
            "1,김천재,100,100,100|2,박수재,95,80,90|3,이자바,80,90,90";
        StringTokenizer st = new StringTokenizer(source, "|");

        while(st.hasMoreTokens()) {
            String token = st.nextToken();

            StringTokenizer st2 = new StringTokenizer(token, ",");
            while(st2.hasMoreTokens()) {
                System.out.println(st2.nextToken());
            }
            System.out.println("------");
        }
    } // main
}
```

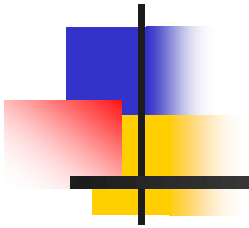
예제3

- split() 는 빈 문자열도 토큰으로 인식하는 반면
- StringTokenizer 는 빈 문자열을 토큰으로 인식하지 않기 때문에 인식하는 토큰의 개수가 서로 다름

- split()는 데이터를 토큰으로 잘라낸 결과를 배열에 담아서 반환하기 때문에 데이터를 토큰으로 바로바로 잘라서 반환하는 StringTokenizer 보다 성능이 떨어짐
- 그러나 데이터의 양이 많은 경우가 아니라면 별 문제가 되지 않음

```
100 |  |  | 200 | 300 | 개수:5
100 | 200 | 300 | 개수:3
```

```
import java.util.*;
class StringTokenizerEx5 {
	public static void main(String[] args) {
		String data = "100,,,200,300";

		String[] result = data.split(",");
		for(int i=0; i < result.length;i++)
			System.out.print(result[i]+" | ");
		System.out.println("개수:"+result.length);

		StringTokenizer st = new StringTokenizer(data, ",");
		int i=0;
		for(;st.hasMoreTokens();i++)
			System.out.print(st.nextToken()+" | ");

		System.out.println("개수:"+i);
	} // main
}
/*
100| | |200|300|개수:5  <-- split()사용결과
100|200|300|개수:3    <-- StringTokenizer사용결과*/
```



Object 클래스

Object 클래스

```
public class ObjectTest3 {
        public static void main(String[] ar) {
                System.out.println("Java!!");
        }
}
```

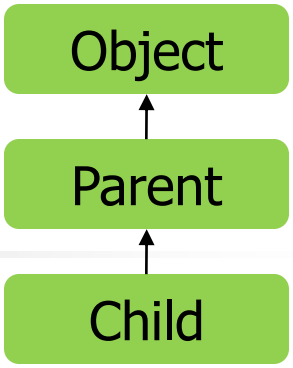
- Object 클래스
  - 모든 클래스의 최고 조상
  - Object 클래스의 멤버들은 모든 클래스에서 바로 사용 가능함
  - 8개의 메서드만 가지고 있음

- 자바에서는 기본적으로 아무런 상속을 받지 않으면 Object 클래스를 상속받음
  - javap 프로그램을 사용하여 자바 클래스가 어떻게 구성되어 있는지를 확인

```
D:₩temp>javap ObjectTest3
Compiled from "ObjectTest3.java"
public class ObjectTest3 extends java.lang.Object{
    public ObjectTest3();
    public static void main(java.lang.String[]);
}
```

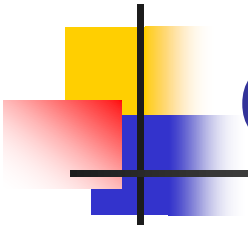
클래스 선언문, 생성자, 메서드들의 목록이 나열됨

Object 클래스



- 왜 모든 클래스는 Object 클래스를 상속받을까?
  - Object 클래스에 있는 메서드들을 통해서 클래스의 기본적인 행동을 정의할 수 있기 때문
  - '사람'은 걷고, 말하고, 생각한다

    이와 마찬가지로, 클래스라면 '이 정도의 메서드는 정의되어 있어야 하고, 처리해주어야 한다'는 것을 정의하는 작업이 필요하기 때문



Object 클래스에서 제공하는 메서드

- Object 클래스에 선언되어 있는 메서드
  - [1] 객체를 처리하기 위한 메서드
  - [2] 쓰레드를 위한 메서드

객체를 처리하기 위한 메서드

메소드	설명
`protected Object clone()`	객체의 복사본을 만들어 리턴한다.
`public boolean equals(Object obj)`	현재 객체와 매개 변수로 넘겨받은 객체가 같은지 확인한다. 같으면 true를 다르면 false를 리턴한다.
`protected void finalize()`	현재 객체가 더 이상 쓸모가 없어졌을 때 가비지 컬렉터(garbage collector)에 의해서 이 메소드가 호출된다.
`public Class<?> getClass()`	현재 객체의 Class 클래스의 객체를 리턴한다.
`public int hashCode()`	객체에 대한 해시 코드(hash code) 값을 리턴한다. 해시 코드라는 것은 "16진수로 제공되는 객체의 메모리 주소"를 말한다.
`public String toString()`	객체를 문자열로 표현하는 값을 리턴한다.

쓰레드를 위한 메서드

메소드	설명
public void notify()	이 객체의 모니터에 대기하고 있는 단일 쓰레드를 깨운다.
public void notifyAll()	이 객체의 모니터에 대기하고 있는 모든 쓰레드를 깨운다.
public void wait()	다른 쓰레드가 현재 객체에 대한 notify() 메소드나 notifyAll() 메소드를 호출할 때까지 현재 쓰레드가 대기하고 있도록 한다.
public void wait(long timeout)	wait() 메소드와 동일한 기능을 제공하며, 매개 변수에 지정한 시간만큼만 대기한다. 즉, 매개 변수 시간을 넘어 섰을 때에는 현재 쓰레드는 다시 깨어 난다. 여기서의 시간은 밀리초로 1/1,000 초 단위다. 만약 1초간 기다리게 할 경우에는 1000을 매개 변수로 넘겨주면 된다.
public void wait(long timeout, int nanos)	wait() 메소드와 동일한 기능을 제공한다. 하지만, wait(timeout)에서 밀리초 단위의 대기시간을 기다린다면, 이 메소드는 보다 자세한 밀리초+나노초(1/1,000,000,000초) 만큼만 대기한다. 뒤에 있는 나노초의 값은 0~999,999 사이의 값만 지정할 수 있다.

Object 클래스의 메서드

```
p2 = p;
if (p.equals(p2)) {
        System.out.println("p와 p2는 같다.");
} else {
        System.out.println("p와 p2는 다르다.");
}
```

```
class Person {
    public void display(){
        System.out.println("this ? " + this);
    }
}
public class ObjectTest {
    public static void main(String[] ar) {
        Person p = new Person();   Person p2 = new Person();
        System.out.println("두 객체가 같나? " + p.equals(p2));
        System.out.println("p객체의 클래스? " + p.getClass());
        System.out.println("p객체의 hashCode는? " + p.hashCode());
        System.out.println("p객체를 표현하는 기본 문자열은? " + p.toString());
        System.out.println("p객체를 표현하는 기본 문자열 약식은? " + p);
        System.out.println("p객체의 hashCode의 16진수 값은?"+Integer.toHexString(p.hashCode()));
        p.display();

    }
}
```

```
두객체가 같나? false
p객체의 클래스? class Person
p객체의 hashCode는? 1762177173
p객체를 표현하는 기본 문자열은? Person@6908b095
p객체를 표현하는 기본 문자열 약식은? Person@6908b095
p객체의 hashCode의 16진수 값은? 6908b095
this ? Person@6908b095
p와 p2는 같다.
```

- toString() 메서드의 결과
  - 클래스명@16진수 해시코드
  - getClass().getName() + '@' + Integer.toHexString(hashCode())

Person 클래스에는 toString() 메서드가 선언되어 있지 않지만, Object 클래스를 자동으로 상속 받으므로, Object 클래스의 메서드인 toString() 메서드를 사용할 수 있다

public void println(Object x)

public String toString()

toString() 메서드

- Object클래스의 toString() 메서드
  - 해당 클래스가 어떤 객체인지를 쉽게 나타낼 수 있는 메서드
  - **객체를 문자열로 표현하는 값을 리턴**
- p.toString() 과 p 를 출력한 결과값이 동일
  - 자바에서는 클래스의 멤버가 할당되어 있는 곳의 주소를 숨기려는 속성이 있음
  - 자바에서는 c언어에서의 포인터를 내부 포인터로 바꾸고, 주소를 출력해 볼 수 있는 예약어를 사용하지 못하도록 만들었음
  - => 직접 주소를 출력해보려고 객체를 출력하면 자동으로 출력 형식의 메서드로 연결해 버림
    - 그 메서드가 toString() 메서드
- toString() 메서드가 자동으로 호출되는 경우
  - [1] System.out.println() 메서드에 매개변수로 들어가는 경우
  - [2] 객체에 대하여 더하기(+) 연산을 하는 경우

예제

```
객체 출력 : Person2 [name=null, age=0]
Person2 [name=null, age=0]
```

```
class Person2 {
	private String name;
	private int age;

	//Object 클래스의 toString() 메서드를 오버라이딩
	public String toString() {
		return "Person2 [name=" + name + ", age=" + age + "]";
	}
}

public class ObjectTest2 {
	public static void main(String[] ar) {
		Person2 p = new Person2();
		System.out.println("객체 출력 : " + p);
		System.out.println(p.toString());
	}
}
```

equals () 메서드

- 연산자 ==
  - 기본 자료형에서는 값이 같은지 비교
  - 참조 자료형에서는 주소값을 비교
- String 클래스의 equals() 메서드
  - 값이 같은지 비교
  - Object 클래스의 equals() 메서드를 오버라이딩하여 문자열 값을 비교하도록 한 것
- Object 클래스의 equals()
  - 매개변수로 객체의 참조변수를 받아서 비교하여 그 결과를 boolean 으로 알려주는 역할
  - 두 객체의 같고 다름을 참조변수의 값으로 판단
  - 주소값을 비교

• Object 클래스에 정의되어 있는 equals 메서드의 실제 내용

```
public boolean equals(Object obj){
        return (this==obj);  //주소값 비교
}
```

예제

```
v1과 v2는 다르다.
v1과 v2는 같다.
```

```
class Test {
	private int value;
	Test(int value) {
		this.value = value;
	}
}
class EqualsEx1{
	public static void main(String[] args) {
		Test v1 = new Test(10);
		Test v2 = new Test(10);
		if (v1.equals(v2)) {  //주소값 비교
			System.out.println("v1과 v2는 같다.");
		} else {
			System.out.println("v1과 v2는 다르다.");
		}
		v2 = v1;
		if (v1.equals(v2)) {
			System.out.println("v1과 v2는 같다.");
		} else {
			System.out.println("v1과 v2는 다르다.");
		}
	}
}
```

```
class Person {
    private long id;
    //equals 오버라이딩
    public boolean equals(Object obj) {
        if(obj!=null && obj instanceof Person) {
            return id ==((Person)obj).id; // obj가 Object타입이므로 id값을 참조하기 위해서는
    Person타입으로 형변환이 필요하다.
        } else {
            return false; // 타입이 Person이 아니면 값을 비교할 필요도 없다.
        }
    }

    Person(long id) {
        this.id = id;
    }
}
class EqualsEx2 {
    public static void main(String[] args){
        Person p1 = new Person(9011081111222L);
        Person p2 = new Person(9011081111222L);

        if(p1==p2) {
            System.out.println("p1과 p2는 같은 주소입니다.");
        } else {
            System.out.println("p1과 p2는 다른 주소입니다.");
        }
```

```
p1과 p2는 다른 주소입니다.
p1과 p2는 같은 사람입니다.
```

```
        if(p1.equals(p2)) {
            System.out.println("p1과 p2는 같은 사람입니다.");
        } else {
            System.out.println("p1과 p2는 다른 사람입니다.");
        }
    }
}
```

hashCode() 메서드

- hashCode()
  - 해싱기법에 사용되는 해시함수를 구현한 것
  - 해싱 – 데이터관리기법 중의 하나인데, 다량의 데이터를 저장하고 검색하는 데 유용함

- Object 클래스의 hashCode() 메서드는 객체의 주소값을 이용해서 해시코드를 만들어 반환
- 해시코드 – 인스턴스의 주소와 관련된 정수값으로
  서로 다른 인스턴스는 서로 다른 해시코드값을 가짐
- hashCode() 메서드
  - 객체의 메모리 주소를 16진수로 리턴함

예제

• String 클래스는 문자열의 내용이 같으면, 동일한 해시코드를 반환하도록 hashCode 메서드를 오버라이딩하였다

• System.identityHashCode(Object x) - Object 클래스의 hashCode 메서드처럼 객체의 주소값으로 해시코드를 생성
=> 모든 객체에 대해 항상 다른 해시코드값을 반환함

```
class Person
{
}
class HashCodeEx1 {
	public static void main(String[] args) {
		String str1=new String("abc");
		String str2=new String("abc");

		//String str1="abc";
		//String str2="abc";

		System.out.println(str1.hashCode());
		System.out.println(str2.hashCode());  //str1, str2는 문자열의 내용이 같으므로 동일한 해시코드 값을 얻는다

		System.out.println(System.identityHashCode(str1));
		System.out.println(System.identityHashCode(str2)); //객체의 주소값 리턴=> str1, str2는 다른 해시코드값을 갖는다
```

```
96354
96354
1839060256
353591321
```

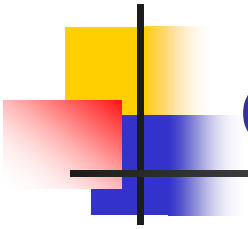
예제

```
        Person p1 = new Person();
        Person p2 = new Person();

        System.out.println("₩n======Person=========");
        System.out.println(p1.hashCode());
        System.out.println(p2.hashCode());
        System.out.println(System.identityHashCode(p1));
        System.out.println(System.identityHashCode(p2));

        Integer n1=10, n2=20;
        System.out.println("₩n====== Integer =========");
        System.out.println(n1.hashCode());
        System.out.println(n2.hashCode());
        System.out.println(System.identityHashCode(n1));
        System.out.println(System.identityHashCode(n2));

    }
}
```

```
96354
96354
1839060256
353591321

======Person=========
722080798
581882789
722080798
581882789

======Integer=========
10
20
562832993
2016131963
```

clone() 메서드

- clone() 메서드
  - **자신을 복제하여 새로운 인스턴스를 생성**하는 일을 함
  - 어떤 인스턴스에 대해 작업을 할 때, 원래의 인스턴스는 보존하고, clone() 메서드를 이용해서 새로운 인스턴스를 생성하여 작업을 하면 작업이전의 값이 보존되므로 작업에 실패해서 원래의 상태로 되돌리거나 변경되기 전의 값을 참고하는 데 도움이 됨
- Object 클래스에 정의된 clone 메서드는 단순히 멤버변수의 값만을 복사함
  - 배열이나 인스턴스가 멤버로 정의되어 있는 클래스의 인스턴스는 완전한 복제가 이루어지지 않음
  - 배열의 경우, 복제된 인스턴스도 같은 배열의 주소를 갖기 때문에 복제된 인스턴스의 작업이 원래의 인스턴스에 영향을 미치게 됨
    - => clone 메서드를 오버라이딩해서 새로운 배열을 생성하고 배열의 내용을 복사하도록 해야 함



clone() 메서드

- Cloneable 인터페이스를 구현한 클래스의 인스턴스만 clone() 을 통한 복제가 가능함
  - 이유 – 인스턴스 복제는 데이터를 복사하는 것이기 때문에 데이터를 보호하기 위해서, 클래스 작성자가 복제를 허용하는 경우, Cloneable 인터페이스를 구현한 경우에만 복제가 가능하도록 하기 위해서임

예제

```
protected   Object   clone()
                throws CloneNotSupportedException
```

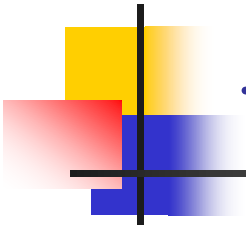
```
x=3, y=5
x=3, y=5
```

```
Point@61de33
Point@14318bb
```

```
class Point implements Cloneable { // Cloneable인터페이스를 구현한 클래스에서만 clone()을 호출할 수 있다. 이 인터페이스를 구현하지 않고 clone()을 호출하면 예외가 발생한다.
    int x;
    int y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public String toString() {
        return "x="+x +", y="+y;
    }

    public Point copy() {
        Object obj=null;
        try {
            obj = clone(); // clone메서드에는 CloneNotSupportedException이 선언되어 있으므로 이 메서드를 호출할 때는 try-catch문을 사용해야한다.
        } catch(CloneNotSupportedException e) {}

        return (Point)obj;
    }
}
```

```
class CloneTest {
    public static void main(String[] args){
        Point original = new Point(3, 5);
        Point copy = original.copy();
        System.out.println(original);
        System.out.println(copy);
    }
}
```

clone() 메서드는 protected 로 선언되어 있음



finalize() 메서드

- finalize() 메서드
  - **인스턴스가 소멸되기 직전에** 자바 가상머신에 의해서 자동으로 호출되는 메서드
  - 인스턴스 소멸 시 반드시 실행되어야 하는 코드가 존재한다면, finalize() 메서드의 활용을 고려할 수 있다

인스턴스 소멸시 반드시 해야 할 일이 있다면 **finalize()** 메서드 이용

```
protected void finalize() throws Throwable
```

예제

```
프로그램을 종료합니다.
인스턴스2이 소멸되었습니다.
인스턴스1이 소멸되었습니다.
```

```
class MyName{
	String objName;
	public MyName(String name){
		objName=name;
	}
	protected void finalize() throws Throwable{  //오버라이딩
		super.finalize();
		System.out.println(objName+"이 소멸되었습니다.");
	}
}
class ObjectFinalize{
	public static void main(String[] args){
		MyName obj1=new MyName("인스턴스1");
		MyName obj2=new MyName("인스턴스2");
		obj1=null;  //생성한 인스턴스는 가비지 컬렉션의 대상이 됨
		obj2=null;
		System.out.println("프로그램을 종료합니다.");
		System.gc();
		System.runFinalization();
	}
}
```

finalize() 메서드

finalize 메서드의 완벽한 호출이 필요한 상황에서는 다음 두 메서드의 연이은 호출이 필요함

System.gc();
System.runFinalization();

- 가비지 컬렉션은 한번도 실행되지 않을 수 있다
  - 빈번한 가비지 컬렉션은 프로그램 성능에 문제를 줄 수 있어서, 특정 알고리즘을 통해서 계산된 시간에 가비지 컬렉션이 수행됨
- 앞 예제는 가비지 컬렉션이 한번도 발생하지 않아서 finalize 메서드가 호출되지 않았다
- System.gc()
  - 명시적으로 가비지 컬렉션을 수행시키는 메서드
  - 이 메서드가 호출되면 자바 가상머신은 가비지 컬렉션을 수행시켜서, 참조되지 않는 인스턴스들을 소멸시킴
  - 하지만 이 메서드만으로는 finalize 메서드의 호출을 100% 보장받지 못함
    - 가비지 컬렉션이 수행되더라도 상황에 따라서 인스턴스의 완전한 소멸은 유보될 수 있기 때문
- System.runFinalization()
  - 완전한 소멸이 유보된 인스턴스들의 finalize 메서드 호출을 위해 System.runFinalization() 메서드를 요청해야 함

static import문

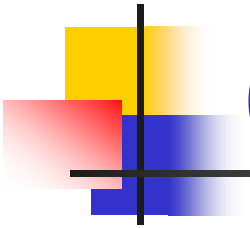
```
import java.util.Date;
import java.util.Scanner;
```

- import 문 사용 – 클래스의 패키지명 생략
- static import 문 사용
  - static 멤버를 호출할 때 클래스명을 생략할 수 있음
  - JDK 5.0 에 추가됨
  - static import 문을 선언할 때는 패키지명도 함께 써주어야 함

```
import static java.lang.System.out;
import static java.lang.Math.*;
```

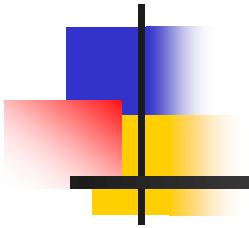
```
System.out.println(Math.random());
```

```
out.println(random());
```

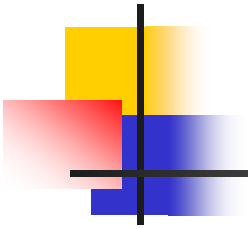


예제

```
import static java.lang.System.out;
import static java.lang.Math.random;
import static java.lang.Math.PI;
//import static java.lang.Math.*;

class StaticImportEx1
{
	public static void main(String[] args)
	{
		// System.out.println(Math.random());
		out.println(random());

		// System.out.println("Mah.PI :"+Math.PI);
		out.println("Mah.PI :"+PI);
	}
}
```



내부 클래스/익명 클래스



내부 클래스(inner class)

- 내부 클래스
  - **클래스 내에 선언된 클래스**
  - 주로 AWT나 Swing과 같은 GUI 어플리케이션의 이벤트 처리에 사용
- 내부 클래스의 장점
  - 한 클래스를 다른 클래스의 내부 클래스로 선언하면 **두 클래스의 멤버들 간에 서로 쉽게 접근**할 수 있다
  - 외부에는 불필요한 클래스를 감춤으로써 코드의 복잡성을 줄일 수 있다

내부 클래스의 종류와 특징

- 변수의 선언위치에 따른 종류와 같다

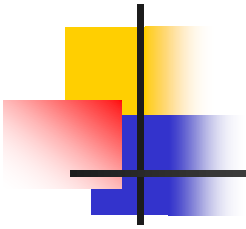
내부 클래스	특 징
인스턴스 클래스 (instance class)	외부 클래스의 멤버변수 선언위치에 선언 외부 클래스의 인스턴스 멤버처럼 다루어짐 주로 외부 클래스의 인스턴스 멤버들과 관련된 작업에 사용될 목적으로 선언됨
스태틱 클래스 (static class)	외부 클래스의 멤버변수 선언위치에 선언 외부 클래스의 static 멤버처럼 다루어짐 주로 외부 클래스의 static 멤버, 특히 static 메서드에서 사용될 목적으로 선언됨
지역 클래스 (local class)	외부 클래스의 메서드나 초기화블럭 안에 선언함 선언된 영역 내부에서만 사용될 수 있음
익명 클래스 (anonymous class)	클래스의 선언과 객체의 생성을 동시에 하는 이름없는 클래스(일회용)

내부 클래스의 선언

- 내부 클래스의 선언위치가 변수의 선언위치와 동일
- 변수가 선언된 위치에 따라 인스턴스 변수, 클래스변수(static 변수), 지역변수로 나뉘듯이 내부 클래스도 이와 마찬가지로 선언된 위치에 따라 나뉜다
- 내부 클래스의 선언위치에 따라 같은 선언위치의 변수와 동일한 유효범위(scope)와 접근성을 갖는다

```
class Outer{
        int iv= 0;
        static int cv=0;
        void myMethod(){
                int lv=0;
        }
}
```

```
class Outer{
        class InstanceInner{}
        static class StaticInner{}
        void myMethod(){
                class LocalInner{}
        }
}
```



내부 클래스의 제어자와 접근성

- 내부 클래스도 클래스이기 때문에 abstract 나 final 과 같은 제어자 사용가능
- private, protected 접근 제어자도 사용가능

예제

내부 클래스중 **static** 클래스만 **static** 멤버를 가질 수 있다

```
100
200
```

```
class InnerEx1 {
    class InstanceInner {
        int iv = 100;
//      static int cv = 100;            // 에러! static변수를 선언할 수 없다.
        final static int CONST = 100;   // static final은 상수이므로 허용한다.

    }

    static class StaticInner {
        int iv = 200;
        static int cv = 200;        // static클래스만 static멤버를 정의할 수 있다.
    }

    void myMethod() {
        class LocalInner {
            int iv = 300;
//              static int cv = 300;            // 에러! static변수를 선언할 수 없다.
            final static int CONST = 300; // static final은 상수이므로 허용
        }
    }

    public static void main(String args[]) {
        System.out.println(InstanceInner.CONST);
        System.out.println(StaticInner.cv);
    }
}
```

예제2

스태틱 클래스는 외부 클래스의 static 멤버만 접근할 수 있다.

외부 클래스의 지역변수는 final이 붙은 변수(상수)만 접근가능하다.
=> JDK 8.0 부터는 지역변수도 접근 가능

```
class InnerEx3 {
	private int outerIv = 0;
	static int outerCv = 0;

	class InstanceInner {
		int iiv = outerIv;  // 외부 클래스의 private멤버도 접근가능하다.
		int iiv2 = outerCv;
	}

	static class StaticInner {
// 스태틱 클래스는 외부 클래스의 static 멤버만 접근할 수 있다.
//		int siv = outerIv;
		static int scv = outerCv;
	}

	void myMethod() {
		int lv = 0;
		final int LV = 0;

		class LocalInner {
			int liv = outerIv;
			int liv2 = outerCv;
//   외부 클래스의 지역변수는 final이 붙은 변수(상수)만 접근가능하다.
//			int liv3 = lv;   //jdk 8.0 부터는 가능
			int liv4 = LV;
		}
	}
}
```

예제3

```
ii.iv : 100
Outer.StaticInner.cv : 300
si.iv : 200
```

```
class Outer {
	class InstanceInner {
		int iv=100;
	}
	static class StaticInner {
		int iv=200;
		static int cv=300;
	}

	void myMethod() {
		class LocalInner {
			int iv=400;
		}
	}
}
class InnerEx4 {
	public static void main(String args[]) {
	 // 인스턴스클래스의 인스턴스를 생성하려면
	 // 외부 클래스의 인스턴스를 먼저 생성해야한다.
		Outer oc = new Outer();
		Outer.InstanceInner ii = oc.new InstanceInner();

		System.out.println("ii.iv : "+ ii.iv);
		System.out.println("Outer.StaticInner.cv : " + Outer.StaticInner.cv);

	  // 스태틱내부 클래스의 인스턴스는 외부 클래스를 먼저 생성하지 않아도 된다.
		Outer.StaticInner si = new Outer.StaticInner();
		System.out.println("si.iv : "+ si.iv);
	}
```

메서드 내에 지역적으로 선언된 내부 클래스는 외부에서 접근할 수 없다.

예제4

```
class Outer {
	int value=10;		// Outer.this.value

	class Inner {
		int value=20;			// this.value
		void method1() {
			int value=30;
			System.out.println("		value :" + value);  //30
			System.out.println("	this.value :" + this.value); //20
			System.out.println("Outer.this.value :" + Outer.this.value); //10
		}
	} // Inner클래스의 끝
} // Outer클래스의 끝

class InnerEx5 {
	public static void main(String args[]) {
		Outer outer = new Outer();
		Outer.Inner inner = outer.new Inner();
		inner.method1();
	}
} // InnerEx5 끝
```

```
            value :30
       this.value :20
Outer.this.value :10
```

예제5

```
class InnerEx2 {
    class InstanceInner {}
    static class StaticInner {}

    // 인스턴스멤버 간에는 서로 직접 접근이 가능하다.
    InstanceInner iv = new InstanceInner();
    // static 멤버 간에는 서로 직접 접근이 가능하다.
    static StaticInner cv = new StaticInner();

    static void staticMethod() {
      // static멤버는 인스턴스멤버에 직접 접근할 수 없다.
//          InstanceInner obj1 = new InstanceInner();
            StaticInner obj2 = new StaticInner();

       //   굳이 접근하려면 아래와 같이 객체를 생성해야한다.
       //   인스턴스클래스는 외부 클래스를 먼저 생성해야만 생성할 수 있다.
            InnerEx2 outer = new InnerEx2();
            InstanceInner obj1 = outer.new InstanceInner();
    }

    void instanceMethod() {
      // 인스턴스메서드에서는 인스턴스멤버와 static멤버 모두 접근 가능하다.
            InstanceInner obj1 = new InstanceInner();
            StaticInner obj2 = new StaticInner();

            //  메서드 내에 지역적으로 선언된 내부 클래스는 외부에서 접근할 수 없다.
//          LocalInner lv = new LocalInner();
    }
```

```
    void myMethod() {
            class LocalInner {}
            LocalInner lv = new LocalInner();
    }
}
```

익명 클래스(anonymous class)

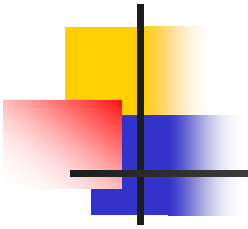
- 익명 클래스
  - 다른 내부 클래스와는 달리 이름이 없다
  - 클래스의 선언과 객체의 생성을 동시에 하기 때문에 단 한번만 사용될 수 있고, 오직 하나의 객체만을 생성할 수 있는 일회용 클래스임

```
class Person{
}

Person p = new Person();
```

```
new  부모클래스이름(){
        //멤버 선언
}
또는

new  구현인터페이스이름(){
        //멤버 선언
}
```

```
class EventHandler implements ActionListener
{
    …
}

EventHandler eh = new EventHandler();
```

```
new ActionListener(){
  ….
};
```



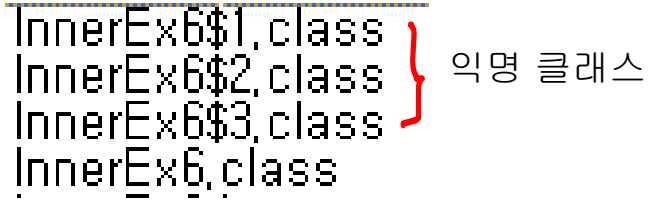
익명 클래스(anonymous class)

- 이름이 없기 때문에 생성자도 가질 수 없으며, 부모 클래스의 이름이나 구현하고자 하는 인터페이스의 이름을 사용해서 정의하기 때문에 하나의 클래스로 상속받는 동시에 인터페이스를 구현하거나 하나 이상의 인터페이스를 구현할 수 없다.
- 오로지 단 하나의 클래스를 상속받거나 단 하나의 인터페이스만을 구현할 수 있다

예제

```
class InnerEx6 {
    Object iv = new Object(){ void method(){} };          // 익명클래스
    static Object cv = new Object(){ void method(){} };   // 익명클래스

    void myMethod() {
        Object lv = new Object(){ void method(){} };      // 익명클래스
    }
}
```

- 컴파일하면 다음 4개의 클래스 파일이 생성됨



- 익명 클래스는 이름이 없기 때문에 '외부 클래스명$숫자.class'의 형식으로 클래스 파일명이 결정됨

예제2- 내부 클래스 이용

```
import java.awt.*;
import java.awt.event.*;

class InnerEx7
{
	public static void main(String[] args)
	{
		Button b = new Button("Start");
		b.addActionListener(new EventHandler());
	}

	static class EventHandler implements ActionListener
	{
	    public void actionPerformed(ActionEvent e) {
		System.out.println("ActionEvent occurred!!!");
	    }
	}
}
```

예제3-익명 클래스 이용

```
import java.awt.*;
import java.awt.event.*;

class InnerEx8
{
    public static void main(String[] args)
    {
        Button b = new Button("Start");
        b.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.out.println("ActionEvent occurred!!!");
                }
            } // 익명 클래스의 끝
        );
    } // main메서드의 끝
} // InnerEx8클래스의 끝
```

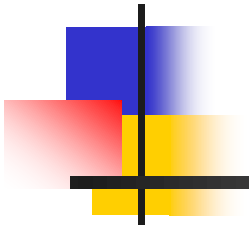
예제4-익명 클래스 이용

```
import java.awt.*;
import java.awt.event.*;
class Anonymous extends Frame{
	Button btn;

	//생성자
	public Anonymous(){
		btn=new Button("닫 기");
		add(btn,"South");
		btn.addActionListener( new ActionListener(){
			public void actionPerformed(ActionEvent e){
				System.exit(0);
			}
		} );
	}

	public static void main(String[] args) {
		Anonymous a=new Anonymous();
		a.setSize(300,300);
		a.setVisible(true);
	}//main
}
```

예제4 – 내부 클래스 이용

```
import java.awt.*;
import java.awt.event.*;
class Anonymous2 extends Frame{
    Button btn;
    public Anonymous2(){
        btn=new Button("닫 기");
        add(btn,"South");
        btn.addActionListener(new EventHandler());
    } //생성자
    public static void main(String[] args){
        Anonymous2 a=new Anonymous2();
        a.setSize(300,300);
        a.setVisible(true);
    }//main

     class EventHandler implements ActionListener{
        public void actionPerformed(ActionEvent e) {
          System.exit(0);
        }
     }//내부 class
}//class
```

```
btn.addActionListener( new ActionListener(){
            public void actionPerformed(ActionEvent e){
                    System.exit(0);
            }
} );
```



과제

과제-369 게임

- [참고]
- int 타입의 변수 num 이 있을 때, 각 자리의 합을 더한 결과를 출력하는 코드를 완성하라. 만일 변수 num의 값이 12345라면, '1+2+3+4+5'의 결과인 15를 출력하라.
  - [주의] 문자열로 변환하지 말고 숫자로만 처리해야 한다.
  - 숫자를 10으로 반복해서 나눠가면서, 10으로 나머지 연산을 하면 일의 자리를 얻어낼 수 있다.
  - 이 값들을 더하기만하면 변수 num에 저장된 숫자의 각 자리수를 모두 더한 값을 구할 수 있다.

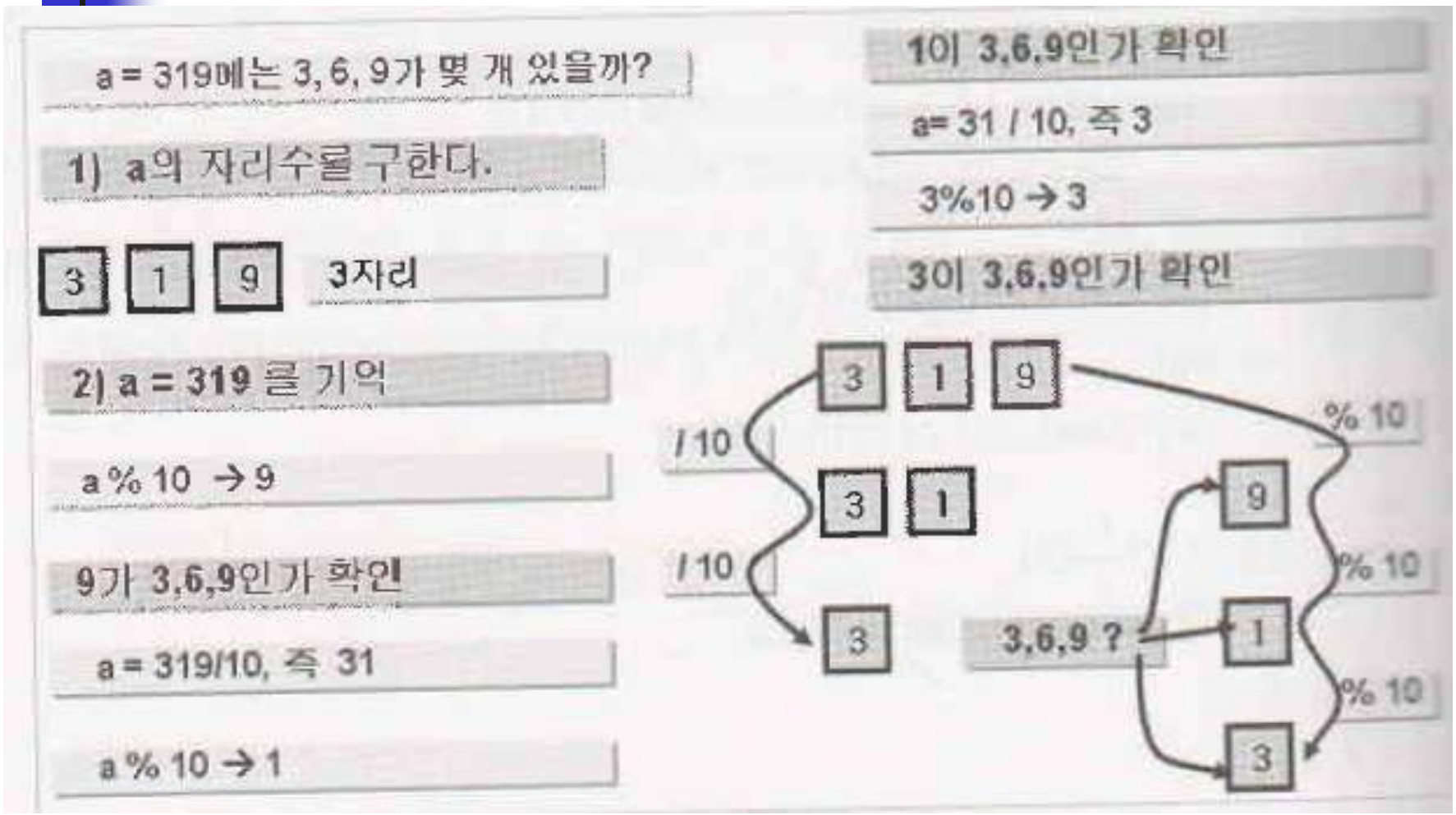
num	num%10
12345	5
1234	4
123	3
12	2
1	1

과제-369 게임

- 각 자리에 3,6,9가 포함하고 있는지 판단하는 369 게임 만들기
  - [힌트]
    - 369 (숫자)의 10으로 나눈 나머지가 3의 배수인지 판단한다
    - num%10, num/10 을 이용하여 숫자의 자리수를 구하고, 그 숫자에 포함된 3,6,9의 개수만큼 * 을 출력한다
    - 3,6,9는 3의 배수이다. 그런데 0%3 도 0 이 되므로 각 자리수의 0은 제외한다
    - 숫자에 3,6,9 중 어느 하나가 1개 이상 포함되면 그 개수만큼 * 을 출력한다. 없다면 해당 숫자를 출력한다.
  - 메서드만 있는 클래스를 만들어서 처리하자

1	2	*	4	5	*	7	8	*	10
11	12	*	14	15	*	17	18	*	20
21	22	*	24	25	*	27	28	*	*
*	*	**	*	*	**	*	*	**	40

과제-369 게임


a = 319에는 3, 6, 9가 몇 개 있을까?
1) a의 자리수를 구한다.
3 1 9
3자리
2) a = 319 를 기억
a % 10 → 9
9가 3,6,9인가 확인
a = 319/10, 즉 31
a % 10 → 1
1이 3,6,9인가 확인
a= 31 / 10, 즉 3
3%10 → 3
3이 3,6,9인가 확인
3 1 9
/ 10
% 10
3 1
/ 10
% 10
3
3,6,9 ?
1
% 10
9
3