마지막 용어는 **싱글톤(Singleton)**이다. 프로그램에서 싱글톤은 클래스의 객체를 공용으로 하나만 사용하려고 할 때 효율적인 방법을 제안한 패턴이다. 이 패턴은 의외로 실무에서 많이 사용한다. 이 패턴을 사용하지 않고 필요한 곳에 매개 변수로 객체를 계속 전달해 줄 수도 있지만 싱글톤으로 클래스를 만들면 더 효율적이라는 이야기다.

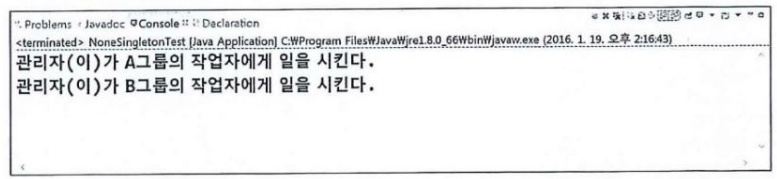
다음 예제는 일반적인 형태로 코드를 작성하여 클래스의 객체를 공용으로 사용하는 경우이다.

예제 NoneSingletonTest.java

```
01  class Master {
02     private String name = "관리자";
03     public String getName() {
04        return name;
05     }
06  }
07  class Worker_A {
08     private Master master;
09     private String name = "A그룹의 작업자";
10     public Worker_A(Master master) {
11        this.master = master;
```

```
12      }
13      public void display() {
14         System.out.println(master.getName() +
              "(이)가 " + name + "에게 일을 시킨다.");
15      }
16   }
17   class Worker_B {
18      private Master master;
19      private String name = "B그룹의 작업자";
20      public Worker_B(Master master) {
21         this.master = master;
22      }
23      public void display() {
24         System.out.println(master.getName() +
              "(이)가 " + name + "에게 일을 시킨다.");
25      }
26   }
27   public class NoneSingletonTest {
28      public static void main(String[] ar) {
29         Master master = new Master();
30         Worker_A wa = new Worker_A(master);
31         Worker_B wb = new Worker_B(master);
32         wa.display();
33         wb.display();
34      }
35   }
```

실행 결과



01 ~ 06:

Worker_A와 Worker_B 클래스의 생성자 메서드에 전달되어 사용될 Master 클래스로 name 필드만 선언되어 있으며 공용으로 사용할 클래스이다.

10 ~ 12:

Master 클래스의 객체를 전달받아 객체를 생성하도록 하는 생성자 메서드이다.

13 ~ 15:

전달받은 Master 클래스의 name 필드 값과 자신의 클래스에 있는 name 필드 값을 출력한다.

17 ~ 26:

Worker_A 클래스와 클래스 이름만 다르고 나머지는 동일하다.

29: Master master = new Master();

공용으로 사용할 Master 클래스의 객체를 생성한다.

30 ~ 31:

Worker_A와 Worker_B 클래스의 객체를 생성하면서 매개 변수로 공용으로 사용할 Master 클래스의 객체를 전달한다.

32 ~ 33:

각 클래스의 정보를 출력한다.

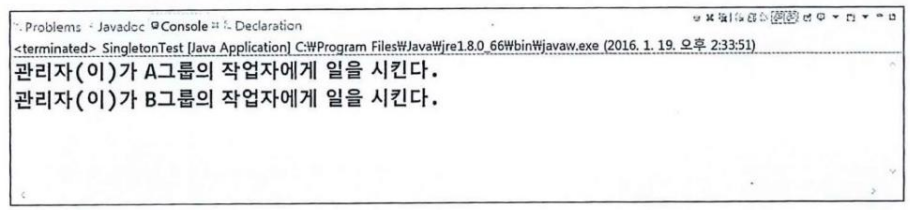
예제를 보면 Master 클래스는 항상 동일한 값을 가지고 있다는 것을 전제로 이 값을 다른 클래스에서 사용할 때 전달하는 방식으로 공용화했다. 그렇다면 싱글톤을 이용하여 코드를 작성하면 어떻게 바뀔까? 다음 예제를 살펴보자.

예제 SngletonTest.java

```
01  class Master_01 {
02     private static Master_01 master = new Master_01();
03     private String name = "관리자";
04     private Master_01() {}
05     public static Master_01 getInstance() {
06        return master;
07     }
08     public String getName() {
09        return name;
10     }
11  }
12  class Worker_A_01 {
13     private String name = "A그룹의 작업자";
14     public void display() {
15        System.out.println(Master_01.getInstance().getName() +
             "(이)가 " + name + "에게 일을 시킨다.");
```

```
16      }
17   }
18   class Worker_B_01 {
19      private String name = "B그룹의 작업자";
20      public void display() {
21         System.out.println(Master_01.getInstance().getName() +
               "(이)가 " + name + "에게 일을 시킨다.");
22      }
23   }
24   public class SingletonTest {
25      public static void main(String[] ar) {
26         Worker_A_01 wa = new Worker_A_01();
27         Worker_B_01 wb = new Worker_B_01();
28         wa.display();
29         wb.display();
30      }
31   }
```

실행 결과



`02: private static Master_01 master = new Master_01();`

자신의 클래스 내에서 공유 필드의 예약어인 static을 사용하여 자신의 객체를 생성한다. 이렇게 생성된 객체는 앞서 static에서 배웠듯이 하나로만 존재하게 된다.

`04: private Master_01() { }`

다른 클래스에서 이 클래스의 객체를 임의로 생성하지 못하도록 생성자 메서드의 접근 제한자를 private으로 선언한다.

`05 ~ 07:`

static 메서드인 getInstance( )를 통해 이미 생성되어 있는 static 필드인 master 객체를 반환한다.

```
15: System.out.println(Master_01.getInstance().getName() +
        "(이)가 " + name + "에게 일을 시킨다.");
```

static 메서드는 클래스 이름으로 접근할 수 있기 때문에 Master_01.getInstance( ) 메서드를 통해 객체를 가져오고 이를 이용하여 public 메서드인 getName( ) 메서드를 호출하여 name 필드 값을 획득한다.

```
26 ~ 27:
```

Master 클래스를 전달하지 않고 Worker_A와 Worker_B 클래스의 객체를 생성한다.

이렇게 싱글톤을 사용하면 매번 공용으로 사용하는 객체를 전달할 필요없이 사용할 수 있다. 대표적인 예로 현재의 날짜를 표시하고자 할 때 사용하는 java.util 패키지의 Calendar 클래스가 있다. 자바 기초를 배우면서 본 적이 있을 것이다.

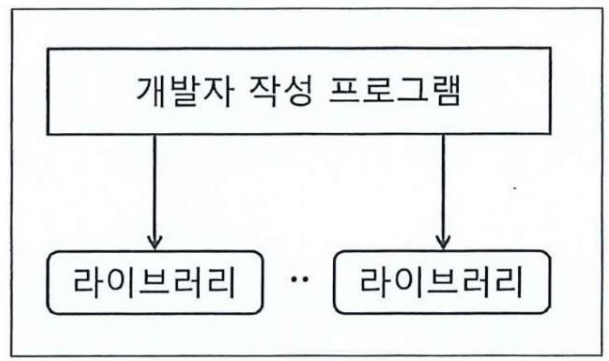
```
Calendar ca = Calendar.getInstance();
```

이처럼 패턴을 사용하면 메모리 활용이나 데이터 관리 등에 있어 좀 더 효율적인 형태의 프로그램을 작성할 수 있다. 따라서 이 책으로 자바의 기초 내용을 배우고서 한 번쯤 디자인 패턴과 관련된 책을 읽어 본다면 많은 도움이 될 것이다.

라이브러리란?

프로그램을 개발할 때 모든 기능을 개발자가 직접 작성하지는 않는다. 필요한 기능을 직접 작성할 수도 있지만 복잡한 로직이 담겨있는 기능의 경우 다른 사람이 작성한 모듈을 가져다 사용할 수 있다. 이때 다른 사람이 작성한 모듈을 라이브러리라고 한다. 혹은 자신이 개발한 클래스들을 라이브러리화할 수도 있다.

개발에서 자기가 직접 작성하는 부분이 갈수록 줄어들고 있다. 외부의 라이브러리를 잘 끌어다 쓰는 것이 생산성 및 프로그램의 품질을 높이는 손쉬운 방법 중의 하나이다.



[도표] 프로그램 및 라이브러리

라이브러리라고 해서 별다른 것은 아니다. 컴파일돼있는 여러 클래스 파일을 확장자가 jar인 파일에 부가 정보와 함께 묶어놓은 형태다.

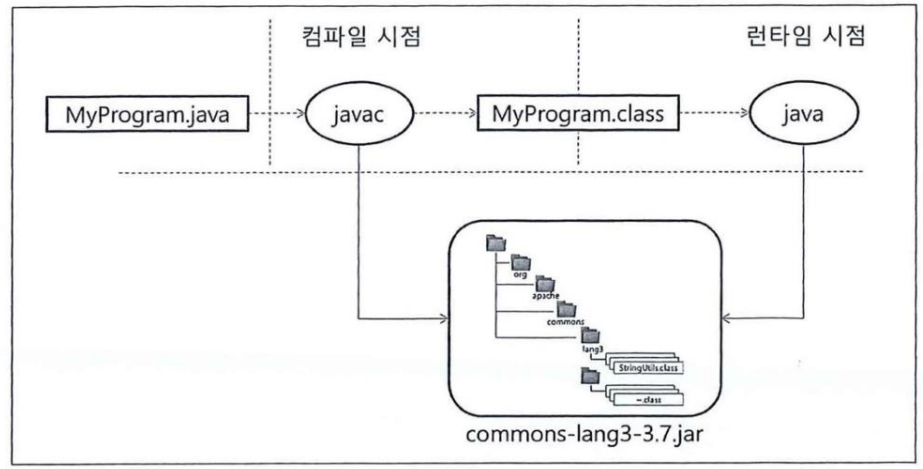
StringUtils라는 클래스가 라이브러리 jar로 제공되고 이를 사용해 다음의 프로그램을 작성한다고 하자.

[예제] 라이브러리 사용 예제 – MyProgram.java

```
package com.javainhand;

import org.apache.commons.lang3.StringUtils;

public class MyProgram {

    public static void main(String[] args) {
        String str = "";
        boolean flag = StringUtils.isEmpty(str);
        System.out.println("StringUtils.isEmpty(str) : "+flag);
    }
}
```

jar 안에는 결국 클래스들이 묶여있다. 위 예에서 org.apache.commons.lang3.StringUtils를 import하고 있는데 이는 실제로 스트링 클래스에 대한 도움 기능을 모아놓은 commons-lang3-3.7.jar라는 라이브러리에 포함돼있다. 해당 라이브러리에 포함돼있는 클래스를 사용하는 코드는 여타 클래스를 사용하는 코드와 동일하다.

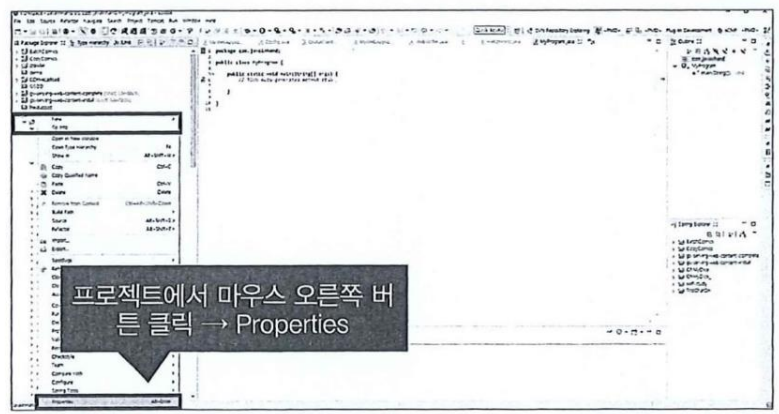
이렇게 jar 형태의 라이브러리를 사용하려면 첫째, 컴파일 타임 때 필요하고, 둘째, 실행 시간에 필요하다. 컴파일 타임 때 필요하다는 것은 javac 명령어를 사용해 확장자가 ~.java 파일을 ~.class로 컴파일할 때 해당 라이브러리를 명시해야 한다는 의미다. 실행 시간에 필요하다는 것은 java 명령어를 사용해 개발자가 작성한 프로그램을 실행할 때 해당 라이브러리를 명시해야 한다는 의미다.

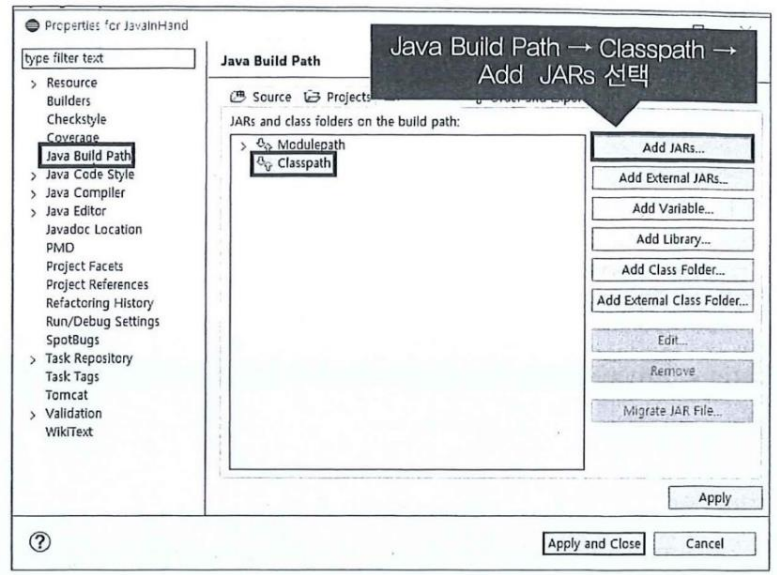


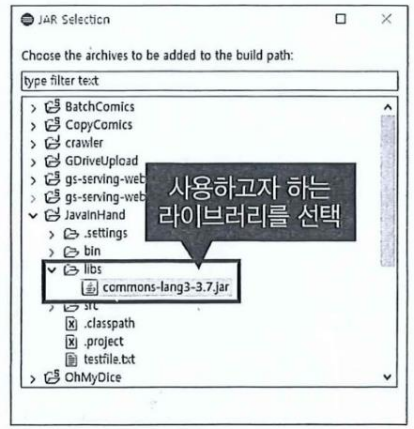
[도표] 라이브러리를 사용한 컴파일 및 실행

앞의 그림에서 알수 있듯이 컴파일 시점에는 컴파일러에게 참조하는 라이브러리를 알려야 하며, 실행 시점에는 가상 머신, 즉 java에게 참조하는 라이브러리를 알려야 한다. 이를 위해 javac와 java의 옵션을 사용해 추가되는 정보를 기술한다.

실무 프로젝트에서는 명령 창에서 javac나 java에 라이브러리 옵션을 주는 방식으로 프로젝트를 진행하지는 않는다. 대신 IDE나 Maven 등의 개발 환경에 라이브러리를 참조하도록 설정한다. 일단 이번에는 이클립스 환경에서 라이브러리를 추가해 보자.







[도표] 라이브러리 추가 작업

사용하려는 라이브러리를 주로 인터넷에서 다운로드해 특정 위치에 복사한 후 앞의 도표와 같은 작업을 수행한다. 앞의 예에서는 프로젝트 폴더 내에 libs라는 디렉터리를 만들고 그 안에 jar 파일을 내려받았다. 이렇게 이클립스 설정을 하면 우리는 이클립스를 통해서 컴파일하고 실행하기 때문에 해당 라이브러리에 대한 참조를 이클립스가 자동으로 수행한다.