자바 프로그래밍

제9장 인터페이스, 람다식, 패키지

학습 내용

학습목차

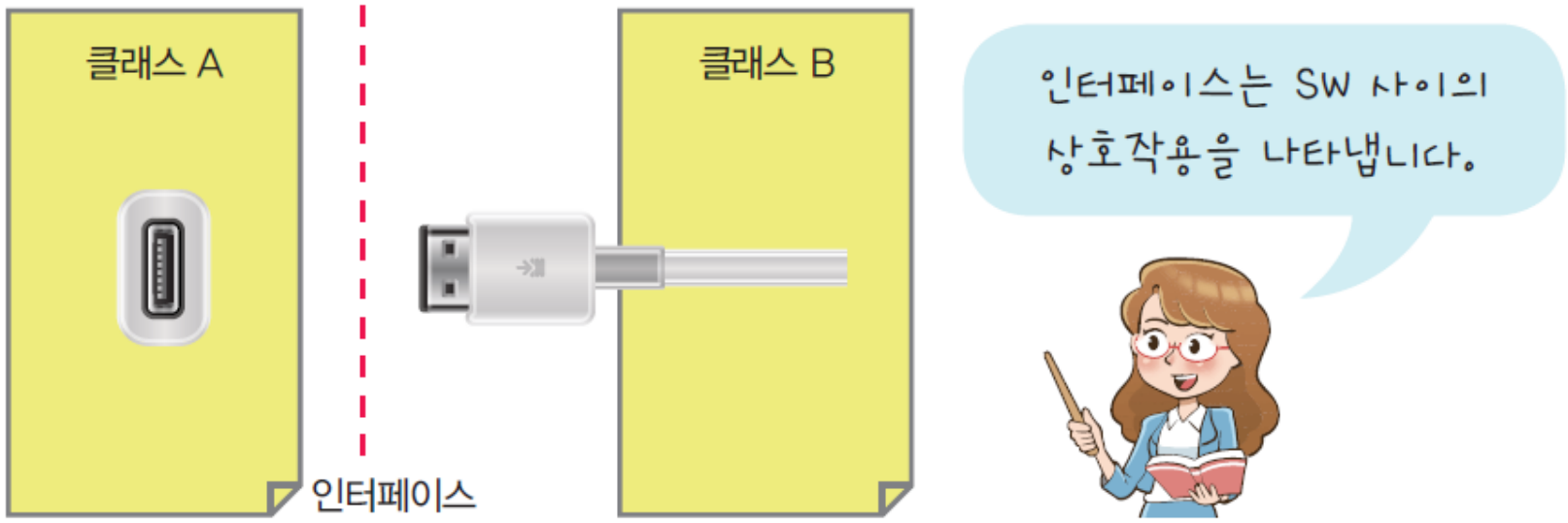
인터페이스

- 인터페이스(interafce)는 서로 다른 장치들이 연결되어 상호 데이터를 주고받는 규격을 의미한다.

USB 인터페이스

자바의 인터페이스(interface)

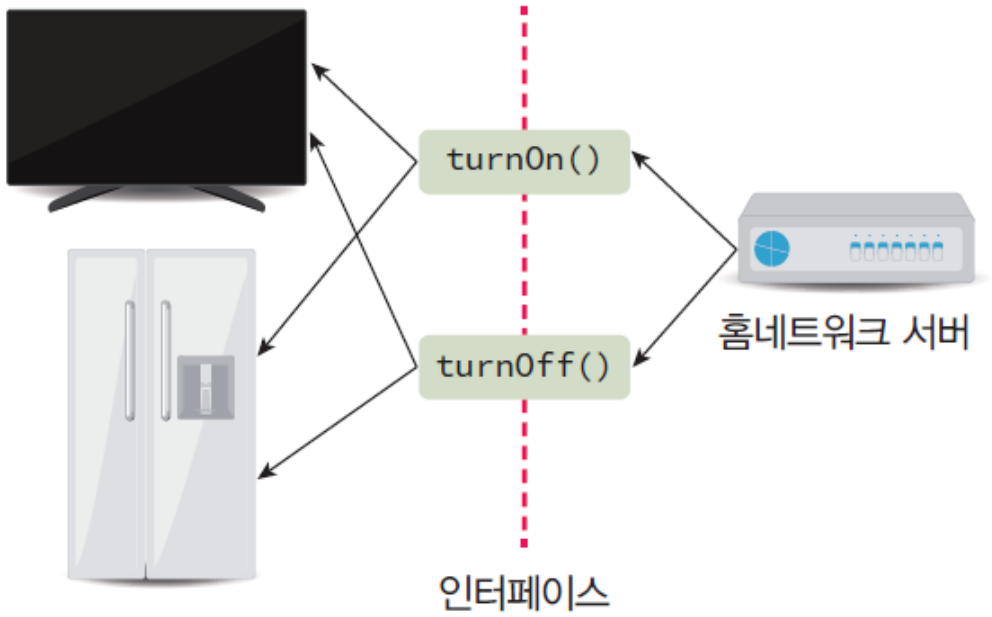
- 클래스와 클래스 사이의 상호 작용의 규격을 나타낸 것이 자바의 인터페이스이다.

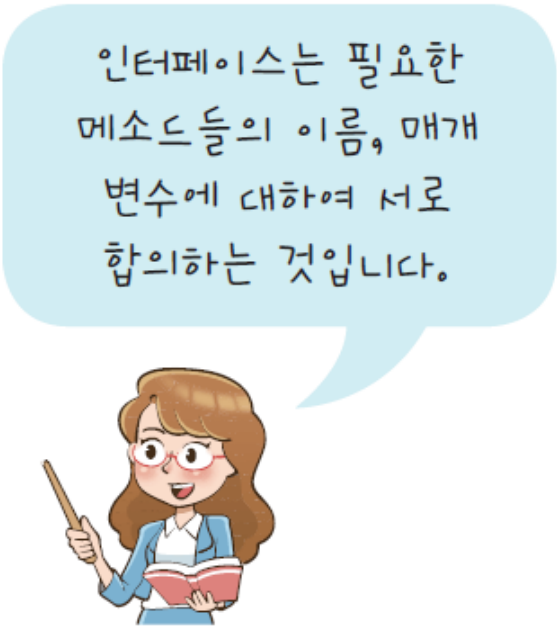


- 인터페이스를 사용하면 구체적인 구현 방법을 제공하지 않으면서도 실행되어야 할 작업을 지정할 수 있다.

인터페이스 예

- 스마트 홈 시스템(Smart Home System)





인터페이스 정의

- 인터페이스 정의 형식

class 대신 키워드 interface 사용

```
public interface 인터페이스이름 {
    반환형    추상메소드1(...) ;
    반환형    추상메소드2(...) ;
    ...
}
```

- 예

```
public interface RemoteControl {
    void turnOn();
    void turnOff();
}
```

인터페이스 정의

- 인터페이스는 추상 메소드들로 이루어진다.
  - 인터페이스 안에서 선언되는 메소드들은 묵시적으로 public abstract ➔ public이나 abstract 수식어 불필요
- 인터페이스는 추상 메소드 외에도 다음은 포함 가능하다.
  - 상수
  - static 메소드
  - default 메소드 ← 이 장 뒤에서 배움
- 인터페이스는 객체를 생성할 수 없다.
- 인터페이스도 extends 키워드를 이용하여 다른 인터페이스를 상속받을 수 있다.

인터페이스 구현

- 인터페이스를 구현하는 클래스를 정의할 수 있다.
  - 구현(implement)이란 인터페이스가 갖는 추상 메소드의 몸체를 정의하는 것
- 인터페이스 구현 형식

```
public class 클래스이름 implements 인터페이스이름 {
    반환형    추상메소드1(...) {
            .....
    }
    반환형    추상메소드2(...) {
            .....
    }
}
```

extends 대신
키워드 implements 사용

홈네트워킹 예제

```
public interface RemoteControl {
    // 2개의 추상 메소드
    void turnOn();          // 켠다.
    void turnOff();         // 끈다.
}
```

인터페이스를 구현

```
public class Television implements RemoteControl {
    boolean onOff = false;
    public void turnOn() {
        onOff = true;       // TV의 전원을 켜기 위한 코드가 들어감
    }
    public void turnOff() {
        onOff = false;      // TV의 전원을 끄기 위한 코드가 들어감
    }
}
```

```
public class TelevisionTest {
    public static void main(String[] args) {
            Television tv = new Television();
            tv.turnOn();
            tv.turnOff();
    }
}
```

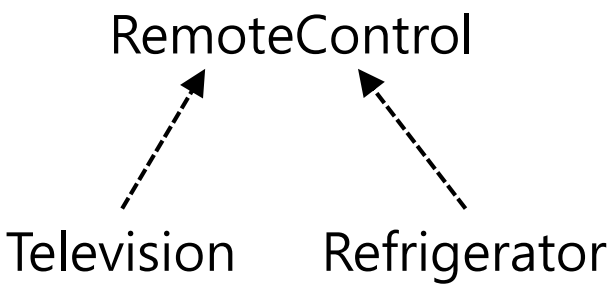
홈네트워킹 예제

```
public interface RemoteControl {
    void turnOn();
    void turnOff();
}
```

```
public class Refrigerator implements RemoteControl {
    …
}
```

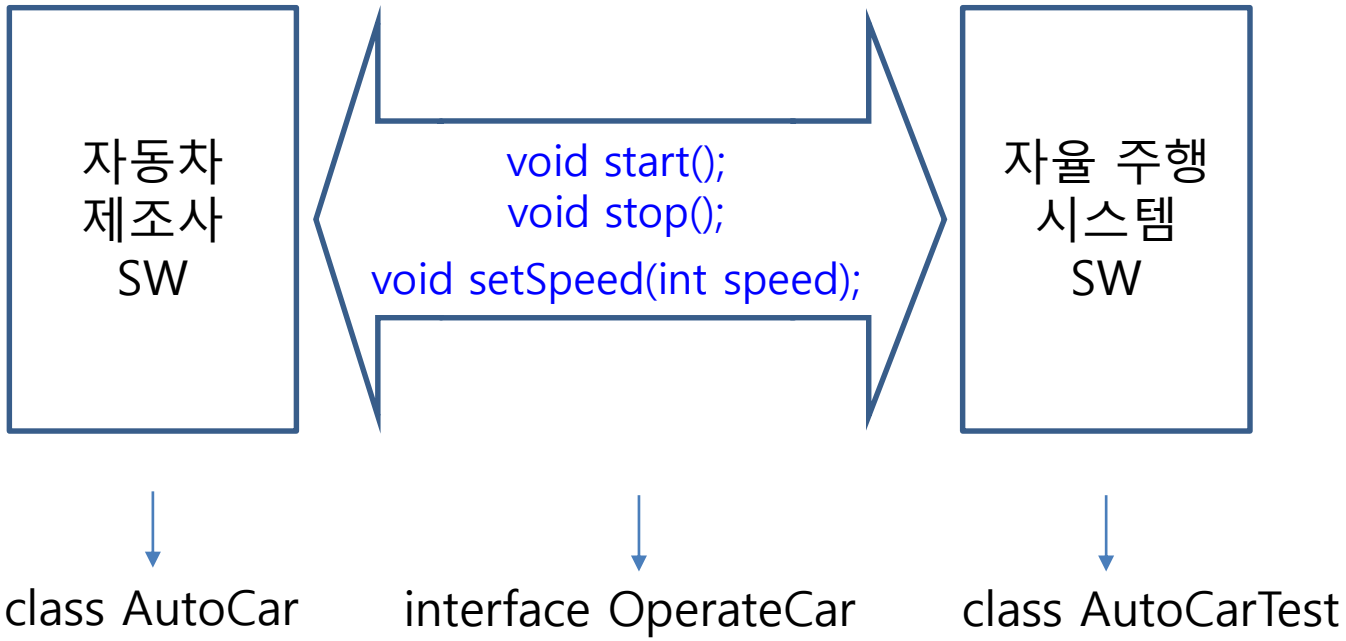
```
public class Television implements RemoteControl {
    …
}
```

```
public class TelevisionTest {
    public static void main(String[] args) {
        Television tv = new Television();
        tv.turnOn();
        tv.turnOff();
        Refrigerator r = new Refrigerator();
        r.turnOn();
        r.turnOff();
    }
}
```



LAB: 자율 주행 자동차

- 추상 메소드를 가지는 인터페이스와 이 인터페이스를 구현하는 클래스를 작성하여 테스트해보자.



```
public class AutoCar implements OperateCar {
    private int speed = 0;
    public void start() {
        speed = 10;
    }
    public void stop() {
        speed = 0;
    }
    public void setSpeed(int speed) {
        this.speed = speed;
    }
    ...
}
```

자동차 제조사 SW

```
public interface OperateCar {
    void start();
    void stop();
    void setSpeed(int speed);
}
```

인터페이스

```
public class AutoCarTest {
    public static void main(String[] args) {
        AutoCar car = new AutoCar();
        car.start();
        car.setSpeed(30);
        car.stop();
    }
}
```

자율 주행 시스템 SW

LAB: 객체 비교하기

- Comparable 인터페이스
  - 객체와 객체의 크기를 비교할 때 사용된다.
  - 이 인터페이스는 우리가 정의하는 것이 아니고 표준 자바 라이브러리에 정의되어 있다.

제네릭은 15장에서 배움

```
public interface Comparable<T> {
    int compareTo(T other);        // 비교 결과(음수, 0, 양수) 리턴
                                   // 예: -1, 0, 1
}
```

```java
public class Rectangle implements Comparable<Rectangle> {

    private int width = 0;
    private int height = 0;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    public int getArea() {
        return width * height;
    }

    @Override
    public int compareTo(Rectangle other) {
        return (getArea() - other.getArea());
    }

    @Override
    public String toString() {
        return "Rectangle[width=" + width + ", height=" + height + "]";
    }
}
```

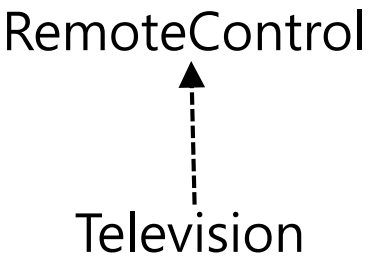
```java
public class RectangleTest {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(2, 3);
        Rectangle r2 = new Rectangle(5, 1);
        int result = r1.compareTo(r2);
        if (result > 0)
            System.out.println(r1 + "가 큼");
        else if (result == 0)
            System.out.println("같음");
        else
            System.out.println(r2 + "가 큼");
    }
}
```

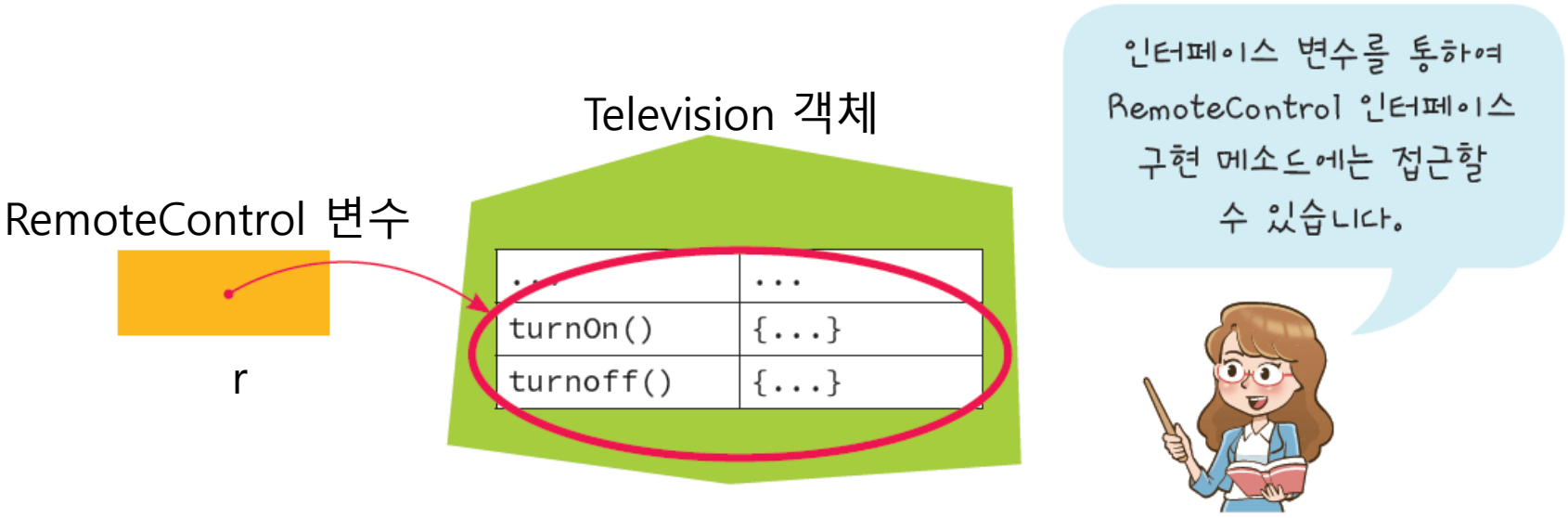
```
Rectangle[width=2, height=3]가 큼
```

인터페이스를 자료형으로 생각하기

- 인터페이스도 하나의 자료형(타입)이다.

```
RemoteControl r = new Television();
r.turnOn();
r.turnOff();
```





```
public interface RemoteControl {
    void turnOn();
    void turnOff();
}
```

```
public class Refrigerator implements RemoteControl {
    ...
}
```

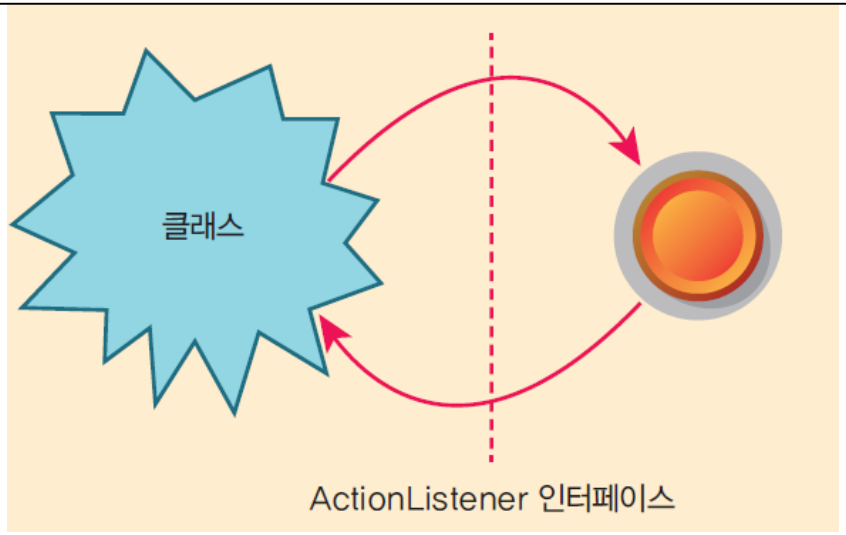
```
public class Television implements RemoteControl {
    ...
}
```

```
public class RectangleTest {
    public static void main(String[] args) {
        Television tv = new Television();
        operate(tv);
        Refrigerator friger = new Refrigerator();
        operate(friger);
    }

    public static void operate(RemoteControl r) {
        r.turnOn();
        r.turnOff();
    }
}
```

예: ActionListener 인터페이스

- 예를 들어 버튼을 눌렀을 때 발생하는 이벤트를 처리하려면 어떤 공통 규격이 있어야 한다.
- ActionListener 인터페이스가 버튼 이벤트를 처리할 때 규격을 정의한다.

```
public interface ActionListener {
        void actionPerformed(ActionEvent event);
}
```



LAB : 타이머 이벤트 처리

- ActionListener는 Timer 이벤트를 처리할 때도 사용된다.
- Timer 클래스는 주어진 시간이 되면 이벤트를 발생시키면서 actionPerformed() 메소드를 호출한다.
- 이 점을 이용하여 1초에 한 번씩 "beep"를 출력하는 프로그램을 작성해보자.
  - ActionListener 인터페이스를 구현한 클래스(MyClass)를 작성하고 객체를 생성하여 Timer에 등록

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;

class MyClass implements ActionListener {
        public void actionPerformed(ActionEvent event) {
                System.out.println("beep");
        }
}

public class CallbackTest {
        public static void main(String[] args) {
                ActionListener listener = new MyClass();
                Timer t = new Timer(1000, listener);
                t.start();
                for (int i = 0; i < 1000; i++) {
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                        }
                }
        }
}
```

일정 시간마다 수행할 작업을 ActionListener 인터페이스의 actionPerformed() 메소드로 정의

스레드는 16장에서 배움

```
beep
beep
beep
...
```

인터페이스 상속하기

- 인터페이스 간에도 상속이 가능하다.

```
public interface RemoteControl {
      void turnOn();  // 켠다.
      void turnOff();  // 끈다.
}
```

(1) 새로운 메소드를 추가하면 RemoteControl을 구현한 모든 클래스들이 동작하지 않는다.

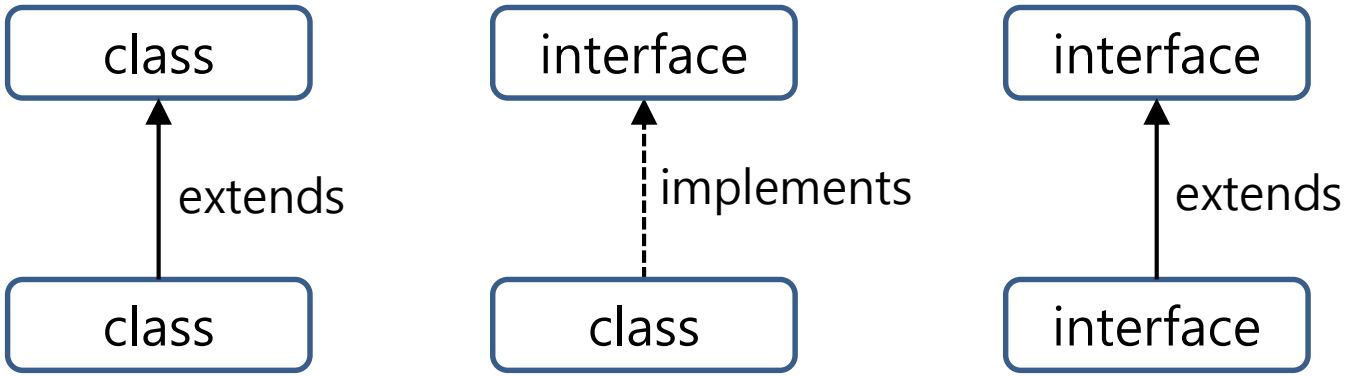
```
public interface RemoteControl {
      void turnOn(); // 켠다.
      void turnoff();  // 끈다.
      void volumeUp(); // 볼륨을 높인다.
      void volumeDown(); // 볼륨을 낮춘다.
}
```

(2) 상속하여 확장하면 그런 문제가 발생하지 않는다.

```
public interface AdvancedRemoteControl extends RemoteControl {
      void volumeUp();        // 볼륨을 높인다.
      void volumeDown();    // 볼륨을 낮춘다.
}
```

인터페이스 상속하기

- 상속과 구현



다중 상속

- 다중 상속(multiple inheritance)은 하나의 클래스가 여러 부모 클래스를 가지는 것이다.
- 예) 하늘을 나는 자동차는 자동차의 특성도 가지고 있지만 비행기의 특성도 가지고 있다.

다중 상속

- 자바에서는 다중 상속을 지원하지 않는다.
- 만일 다중 상속이 허용된다면 문제가 발생한다.

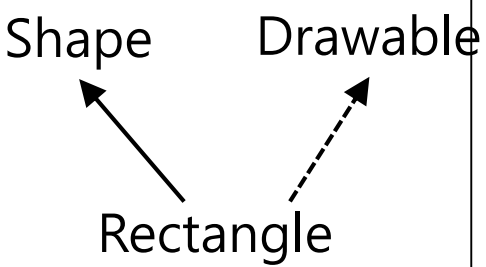
```
class A {
    void print() { System.out.println("A"); }
     …
}
class B {
    void print() { System.out.println("B"); }
     …
}
class Sub extends A, B {     // 만약 다중 상속이 허용된다면
      …
}
```

```
Sub obj = new Sub();
obj.print();                    // 어떤 수퍼 클래스의 print()를 호출하는가?
```

다중 상속

- 인터페이스를 이용하면 다중 상속의 효과를 낼 수 있다.

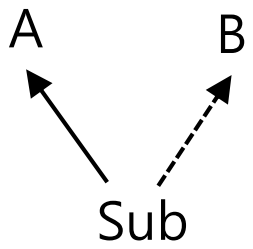
```
class Shape {
    int x, y;
}

interface Drawable {
    void draw();
}

class Rectangle extends Shape implements Drawable {
    int width, height;
    public void draw() {
        System.out.println("Rectangle draw");
    }
}
```



다중 상속

- 앞의 문제도 해결할 수 있다.

```
class A {
    void print() { System.out.println("A"); }
}

interface B {
    void print();
}

class Sub extends A implements B {
}
```




```
Sub obj = new Sub();
obj.print();                    // 출력 A
```

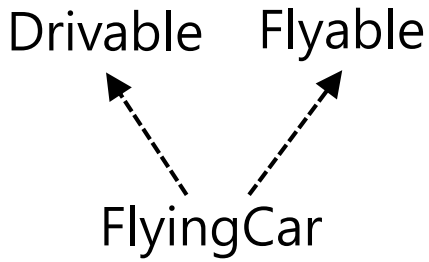
다중 상속

- 한 클래스가 여러 개의 인터페이스를 구현할 수 있다.

```
interface Drivable {
    void drive();
}
```

```
interface Flyable {
    void fly();
}
```

```
public class FlyingCar implements Drivable, Flyable {
    public void drive() {
            System.out.println("I'm driving");
    }
    public void fly() {
            System.out.println("I'm flying");
    }
    public static void main(String[] args) {
            FlyingCar obj = new FlyingCar();
            obj.drive();
            obj.fly();
    }
}
```




```
I'm driving
I'm flying
```

상수 공유

- 인터페이스의 또 다른 용도는 여러 클래스에서 공유하는 상수를 정의하는 것이다.
  - 인터페이스 = 추상 메소드 + 상수 정의

생략해도 public static final임

```
interface Days {
    public static final int SUNDAY = 1, MONDAY = 2,
                           TUESDAY = 3, WEDNESDAY = 4, THURSDAY = 5,
                           FRIDAY = 6,  SATURDAY = 7;
}

public class DayTest implements Days {
    public static void main(String[] args) {
        System.out.println("일요일: " + SUNDAY);
    }
}
```

어떤 인터페이스의 상수를 공유하려면 클래스가 그 인터페이스를 구현하면 된다.

디폴트 메소드(default method)

- 디폴트 메소드는 인터페이스 개발자가 메소드의 디폴트 구현을 제공할 수 있는 기능이다.

```
interface MyInterface {
      void myMethod1();                  // 추상 메소드
      default void myMethod2() {     // 디폴트 메소드
            System.out.println("B");
      }
}
public class DefaultMethodTest implements MyInterface {
      public void myMethod1() {      // 추상 메소드 구현
            System.out.println("A");
      }
      public static void main(String[] args) {
            DefaultMethodTest obj = new DefaultMethodTest();
            obj.myMethod1();
            obj.myMethod2();
      }
}
```

디폴트 메소드는 구현하지 않아도 바로 사용할 수 있다.

```
A
B
```

디폴트 메소드(default method)

- 기존의 코드를 건드리지 않고 인터페이스를 확장할 수 있도록 한다.

```
interface OperateCar {
      void start();
      void stop();
      default void fly() {
            System.out.println("fly");
      }
}
class OldCar implements OperateCar {
      public void start() { }
      public void stop() { }
}
public class DefaultMethodTest {
     public static void main(String[] args) {
            OldCar car = new OldCar();
            car.fly();
     }
}
```

(1) 인터페이스에 새롭게 추가된 메소드 – 디폴트 메소드를 추가했다.

(2) 예전 클래스 구현을 변경하지 않아도 된다. 즉, 인터페이스에 추가된 메소드를 구현하지 않아도 된다.

(3) 예전 클래스 객체에서도 새롭게 추가된 메소드를 호출할 수 있다.

정적 메소드(static method)

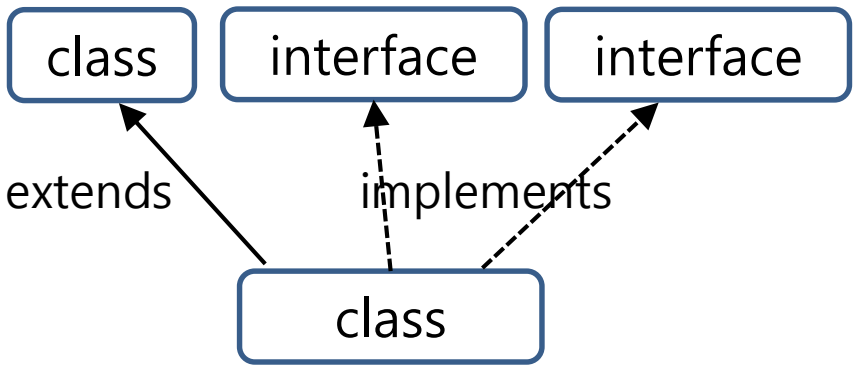
- 인터페이스는 전통적으로 추상적인 규격이기 때문에 정적 메소드를 둘 수 없었지만, 최근에 인터페이스에서도 정적 메소드가 있는 것이 좋다고 간주되고 있다.

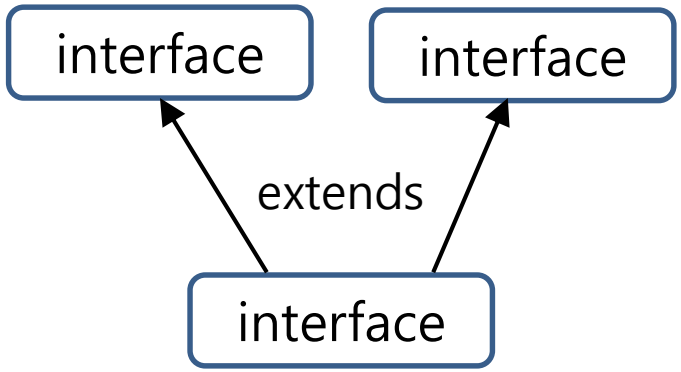
```
interface MyInterface {
    static void print(String msg) { // 정적 메소드
        System.out.println(msg + ": 인터페이스의 정적 메소드 호출");
    }
}
public class StaticMethodTest {
    public static void main(String[] args) {
        MyInterface.print("Java 8");
    }
}
```

```
Java 8: 인터페이스의 정적 메소드 호출
```

인터페이스 vs. 추상클래스

- 공통점
  - 객체 생성 불가능
  - 추상 메소드
- 차이점
  - 추상 클래스에는 인스턴스 변수, 인스턴스 메소드를 둘 수 있지만, 인터페이스에는 불가능
  - 추상 클래스는 클래스이므로 다중 상속 불가능하지만, 인터페이스는 다중 상속 가능





인터페이스 vs. 추상클래스

- 추상 클래스를 이용하는 경우
  - 관련 클래스들 간에 코드를 공유하고자 하는 경우
  - public 이외의 접근 지정자를 사용해야 하는 경우
  - 인스턴스 변수를 두고자 하는 경우
- 인터페이스를 이용하는 경우
  - 관련 없는 클래스들에게 공통의 상호작용 규격(인터페이스)을 제공하고자 하는 경우
  - 다중 상속이 필요한 경우

무명 클래스(anonymous class)

- 무명 클래스는 클래스 몸체는 정의되지만 이름이 없는 클래스이다.
- 무명 클래스는 클래스를 정의하면서 동시에 객체를 생성하게 된다.
- 이름이 없기 때문에 한번만 사용 가능하다.
  - 객체를 하나만 생성하면 되는 경우에 사용

형식

```
new 부모클래스이름()
{
    ...
    // 클래스 몸체
}
```

상속받고자 하는 부모 클래스의 이름이나 구현하고자 하는 인터페이스의 이름을 적어준다.

무명 클래스 예제

```
interface RemoteControl {
        void turnOn();
        void turnOff();
}

public class AnonymousClassTest {
        public static void main(String[] args) {

                RemoteControl tv = new RemoteControl() {
                        public void turnOn() {
                                System.out.println("TV turnOn()");
                        }
                        public void turnOff() {
                                System.out.println("TV turnOff()");
                        }
                };
                tv.turnOn();
                tv.turnOff();
        }
}
```

무명 클래스 정의와
객체 생성을 동시에

```
TV turnOn()
TV turnOff()
```

```
interface RemoteControl {
        void turnOn();
        void turnOff();
}

public class AnonymousClassTest {
       public static void main(String[] args) {

            class Television implements RemoteControl {
                 public void turnOn() {
                        System.out.println("TV turnOn()");
                 }
                 public void turnOff() {
                        System.out.println("TV turnOff()");
                 }
            }

            RemoteControl tv = new Television();
            tv.turnOn();
            tv.turnOff();
       }
}
```

무명 클래스를 사용하지 않고,
이름 있는 클래스를 정의한 경우

```
TV turnOn()
TV turnOff()
```

람다식(lambda expression)

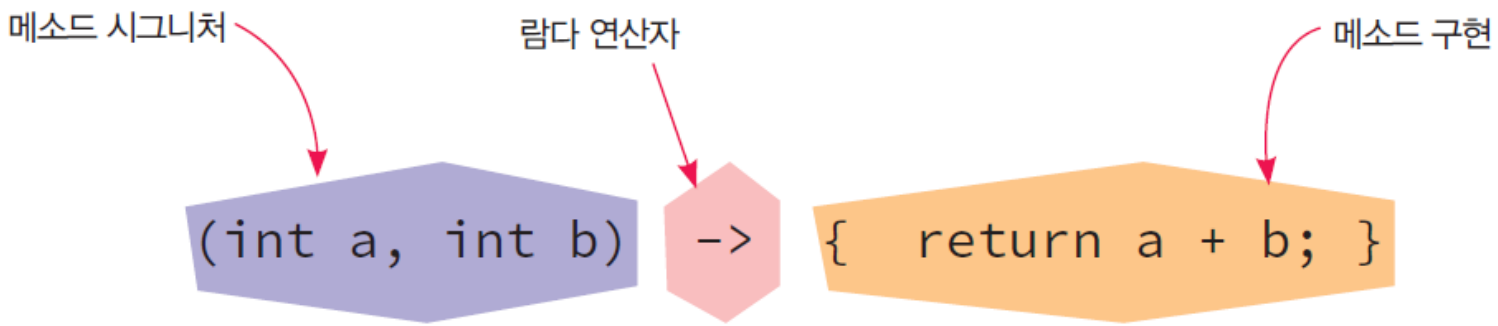
- 람다식은 나중에 실행될 목적으로 다른 곳에 전달될 수 있는 함수이다.
- 람다식은 이름 없는 메소드
- 람다식을 사용하는 이유는 간결함 때문
  - 람다식을 이용하면 메소드가 필요한 곳에 간단히 메소드를 보낼 수 있다.
  - 메소드가 딱 한번만 사용되고 길이가 짧은 경우에 유용하다.

람다식 구문

- 람다식은 (argument) -> (body) 구문을 사용하여 작성

형식

```
(arg1, arg2...)  ->  { body }

(type1 arg1, type2 arg2...)  ->  { body }
```



람다식의 예

```
() -> System.out.println("Hello World")

(String s) -> { System.out.println(s); }

(a, b) -> a + b

(double x) -> x * 100

x -> x * 100

() -> 69

() -> { int x = 69; return x; }
```

람다식 사용 예

```
public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public static int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String[] args) {
        System.out.println(operateBinary(4, 2, (a, b) -> a + b));
     }
}
```

람다식을 이용하면 메소드가 하나인 인터페이스를 간결하게 사용할 수 있다.

```
6
```

람다식을 사용하지 않고,
무명 클래스를 사용한 경우

```
public class Calculator {
    interface IntegerMath {
        int operation(int a, int b);
    }

    public static int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String[] args) {
        System.out.println(operateBinary(4, 2,
                        new IntegerMath() {
                            public int operation(int a, int b) {
                                return a + b;
                            }
                        }
            ));
    }
}
```

```
6
```

함수 인터페이스(functional interface)

- 함수 인터페이스는 single-method interface
  - abstract method를 하나만 갖는 인터페이스
  - default method, static method는 여러 개 가져도 됨
  - 앞의 예에서 본 IntegerMath, 자바의 Runnable, ActionListener 인터페이스가 함수 인터페이스임
- 함수 인터페이스 정의 예

```
interface IntegerMath {
    int operation(int a, int b);
}
```

- 함수 인터페이스를 구현할 때 메소드 이름을 생략해도 어떤 메소드인지 알 수 있다.
  - 인터페이스를 구현하는 클래스를 정의하는 대신, 람다식을 사용하여 단순화할 수 있다.

함수 인터페이스와 람다식

- 함수 인터페이스에 람다식을 대입할 수 있다.
- 예:

```
Runnable r = () -> System.out.println("스레드 실행중") ;
```

- 함수 인터페이스를 지정하지 않으면 컴파일러가 자동으로 형변환한다.

```
new Thread(
        () -> System.out.println("스레드 실행중")
).start();
```

  - 컴파일러가 Thread 클래스의 생성자 public Thread(Runnable r) {...} 을 보고 자동으로 위의 람다식을 Runnable로 형변환

함수 인터페이스와 람다식

- 앞의 람다식 사용 예:

```
System.out.println(operateBinary(4, 2, (a, b) -> a + b) );
```

➔다음과 같이 대입하여 사용 가능

```
IntegerMath addition = (a, b) -> a + b;
System.out.println(operateBinary(4, 2, addition));
```

람다식 대입 예

```
public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public static int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;

        System.out.println(operateBinary(4, 2, addition));
        System.out.println(operateBinary(4, 2, subtraction));
    }
}
```

람다식 대입 예

```
public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public static void main(String... args) {
        IntegerMath addition = (a, b) -> a + b;
        System.out.println(addition.operation(4, 2));
    }
}
```

람다식 사용 예

```
@FunctionalInterface
interface MyInterface {
    void sayHello();
}

public class LambdaTest1 {

    public static void main(String[] args) {
        MyInterface hello = () -> System.out.println("Hello Lambda!");
        hello.sayHello();
    }
}
```

매개변수가 없는 람다식

```
Hello Lambda!
```

람다식 사용 예

```
@FunctionalInterface
interface YourInterface {
    void calculate(int x, int y);
}

public class LambdaTest2 {

    public static void main(String[] args) {
        YourInterface hello = (a, b) -> {
                int result = a * b;
                System.out.println(a + "*" + b);
                System.out.println(result);
        };

        hello.calculate(10, 20);
    }
}
```

매개변수가 2개이고
문장이 여러개인 람다식

```
10*20
200
```

람다식 사용 예

- 무명 클래스를 사용하여 버튼의 클릭 이벤트를 처리하는 코드

```
button.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            System.out.println("버튼이 클릭되었음!");
        }
});
```

하나의 메소드를 구현한 클래스 객체 생성하여 전달했음

실제 필요한 것은 하나의 메소드 전달이므로, 객체 생성 없이 람다식 전달로도 충분

- 람다식을 사용하여 위와 동일한 작업을 수행하는 코드

```
button.addActionListener( (e) -> {
        System.out.println("버튼이 클릭되었음!");
});
```

람다식 사용 예

- 무명 클래스를 사용하여 스레드를 실행하는 코드

```
new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("스레드 실행!");
        }
}).start();
```

- 람다식을 사용하여 위와 동일한 작업을 수행하는 코드

```
new Thread(
        () -> System.out.println("스레드 실행!")
).start();
```

LAB: 타이머 이벤트 처리

- 앞에서 Timer 클래스를 사용하여서 1초에 한 번씩 “beep”를 출력하는 프로그램을 작성한 바 있다. 람다식을 이용하면 얼마나 간결해지는 지를 확인하자.

```
beep
beep
beep
...
```

앞에서 작성한 SOLUTION

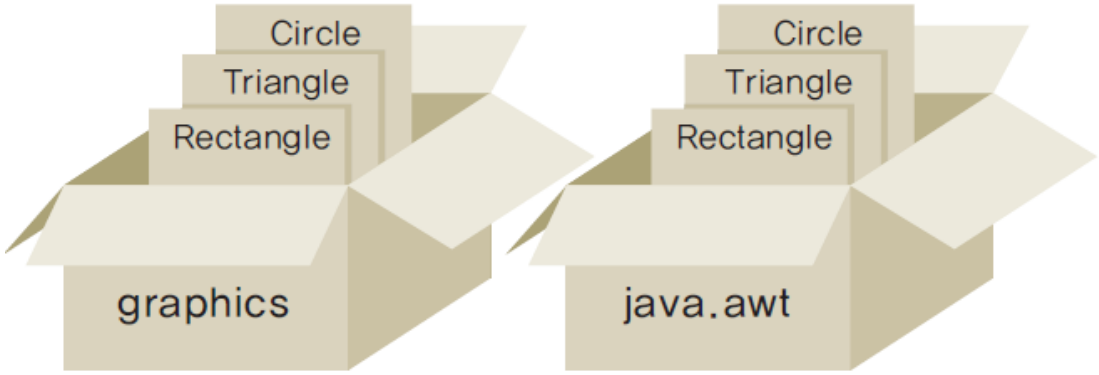
```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;

class MyClass implements ActionListener {
        public void actionPerformed(ActionEvent event) {
                System.out.println("beep");
        }
}

public class CallbackTest {
        public static void main(String[] args) {
                ActionListener listener = new MyClass();
                Timer t = new Timer(1000, listener);
                t.start();
                for (int i = 0; i < 1000; i++) {
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                        }
                }
        }
}
```

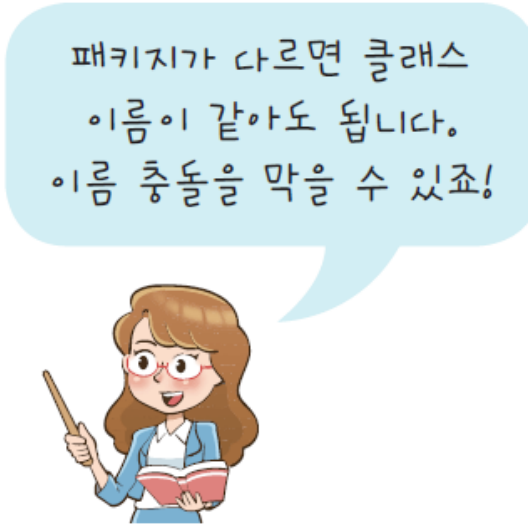
람다식을 이용한 SOLUTION

```
import javax.swing.Timer;

public class CallbackTest {

    public static void main(String[] args) {
            Timer t = new Timer(1000, event -> System.out.println("beep"));
            t.start();

            for (int i = 0; i < 1000; i++) {
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
    }
}
```

패키지(package)

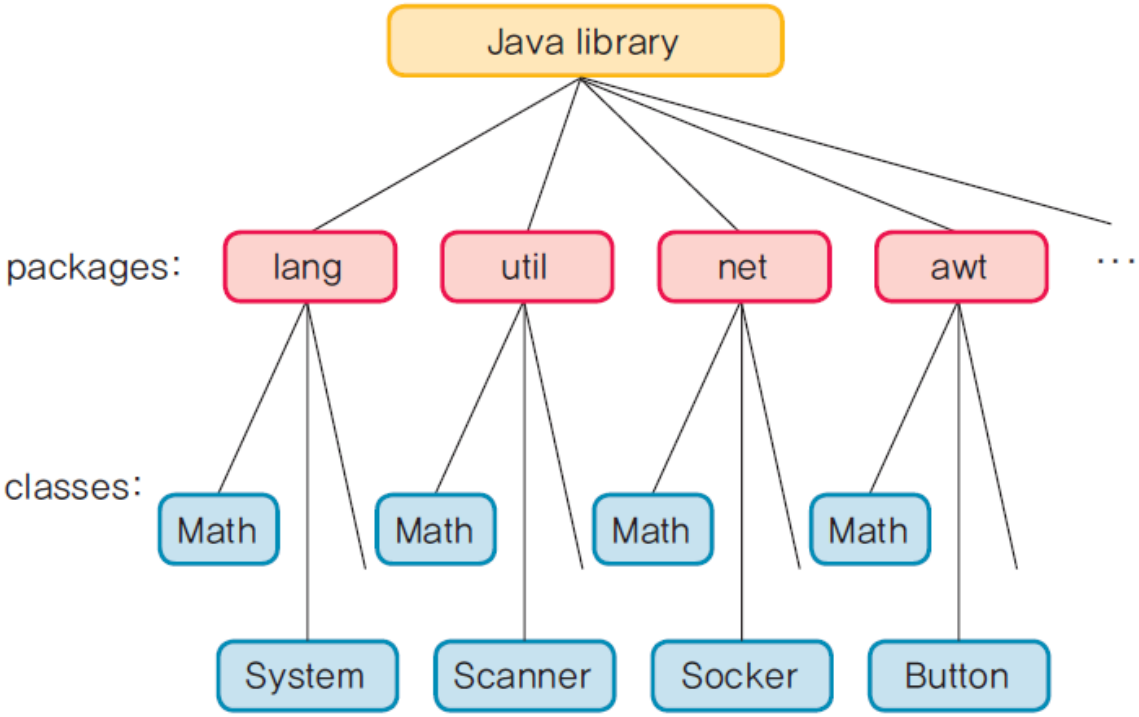
- 패키지는 서로 관련 있는 클래스나 인터페이스들을 하나로 묶은 것이다.

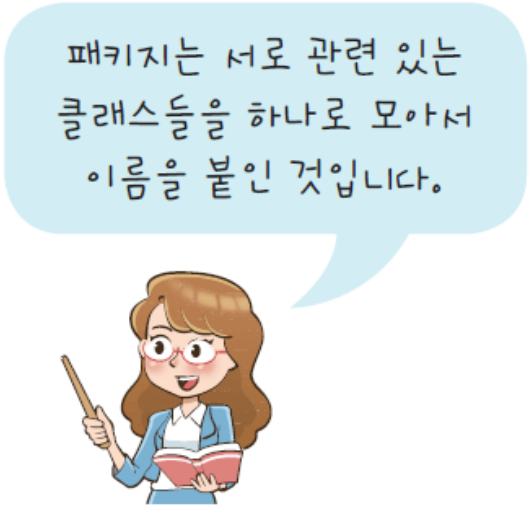




자바 라이브러리

- 자바가 제공하는 라이브러리도 기능별로 패키지로 묶여서 제공되고 있다.





패키지를 사용하는 이유

- 패키지를 이용하면 서로 관련된 클래스들을 하나의 단위로 모을 수 있다.
- 패키지를 이용하여 더욱 세밀한 접근 제어를 구현할 수 있다.
  - 클래스들을 패키지 안에서만 사용 가능하게 선언할 수 있다.
- 패키지를 사용하는 가장 중요한 이유는 "이름공간(name space)" 때문이다.
  - 모든 클래스는 서로 다른 이름을 가져야 한다.
  - 패키지가 다르면 개발자들은 동일한 클래스 이름을 마음 놓고 사용할 수 있다.
  - 예: java.util 패키지의 Date 클래스 → java.util.Date
    java.sql 패키지의 Date 클래스 → java.sql.Date

패키지 정의

```
package library;
public class Circle
{
    . . .
}
```

Circle.java

```
package library;
public class Rectangle
{
    . . .
}
```

Rectangle.java

패키지 이름

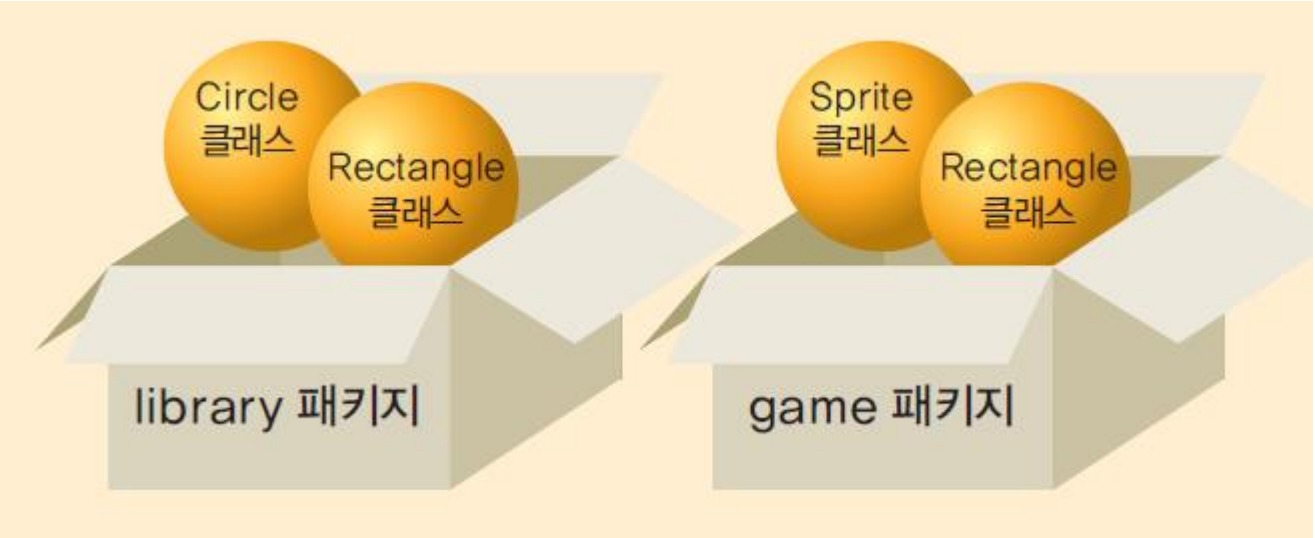
- 패키지 이름이 중복되지 않도록 결정하는 규칙
  - 패키지 이름은 일반적으로 소문자만 사용
  - 인터넷 도메인 이름의 역순을 사용
    - 예를 들어 kr.co.company.library
  - 자바 언어 자체의 패키지는 java 또는 javax로 시작
- 디폴트 패키지(default package)
  - 이름 없는 패키지
  - 패키지 이름을 지정하지 않으면 디폴트 패키지에 속함
  - 임시적인 프로그램 작성할 경우에만 사용

자바에서 지원하는 패키지

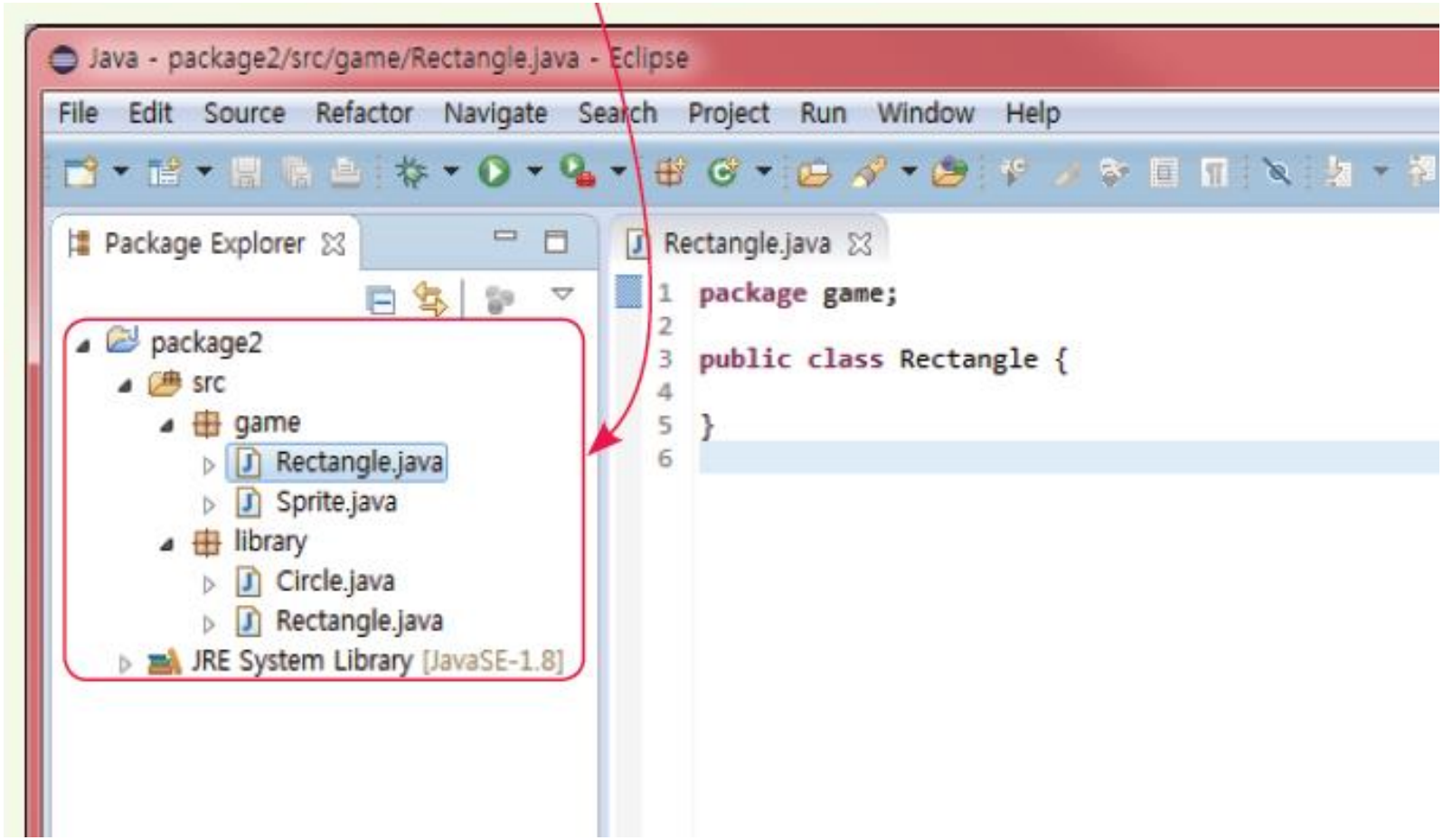
패키지	설명
java.applet	애플릿을 생성하는데 필요한 클래스
java.awt	그래픽과 이미지를 위한 클래스
java.beans	자바빈즈 구조에 기초한 컴포넌트를 개발하는데 필요한 클래스
java.io	입력과 출력 스트림을 위한 클래스
java.lang	자바 프로그래밍 언어에 필수적인 클래스
java.math	수학에 관련된 클래스
java.net	네트워킹 클래스
java.nio	새로운 네트워킹 클래스
java.rmi	원격 메소드 호출(RMI) 관련 클래스
java.security	보안 프레임워크를 위한 클래스와 인터페이스
java.sql	데이터베이스에 저장된 데이터를 접근하기 위한 클래스
java.util	날짜, 난수 생성기 등의 유틸리티 클래스
javax.imageio	자바 이미지 I/O API
javax.net	네트워킹 애플리케이션을 위한 클래스
javax.swing	스윙 컴포넌트를 위한 클래스
javax.xml	XML을 지원하는 패키지

LAB: 패키지 생성하기

- 예를 들어서 어떤 회사에서 게임을 개발하려면 library 팀과 game 팀의 소스를 합쳐야 한다고 가정해보자.



SOLUTION


Java - package2/src/game/Rectangle.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
package2
src
game
Rectangle.java
Sprite.java
library
Circle.java
Rectangle.java
JRE System Library [JavaSE-1.8]
Rectangle.java
1 package game;
2
3 public class Rectangle {
4
5 }
6

패키지의 사용

- 특정 패키지 내 구성 요소를 참조하는 방법

  (1) 패키지 이름(경로)까지 포함하는 완전한 이름으로 참조

  ```
  java.util.Scanner scan = new java.util.Scanner(System.in);
  ```

  (2) 원하는 패키지 구성 요소만 import

  ```
  import  java.util.Scanner;
  …
  Scanner scan = new Scanner(System.in);
  ```

  (3) 패키지 전체를 import

  ```
  import java.util.*;
  ```

- java.lang 패키지는 자동으로 import됨
  - 예를 들어 다음 문장은 불필요

  ```
  import java.lang.String;          // 불필요
  ```

정적 import 문장

- 클래스 안에 정의된 정적 상수나 정적 메소드를 사용하는 경우에 정적 import 문장을 사용하면 클래스 이름을 생략해도 된다.
  - 예:

```
double r = Math.cos(Math.PI * theta);
```

➔ 정적 import 문을 사용하면 Math 생략 가능하다.

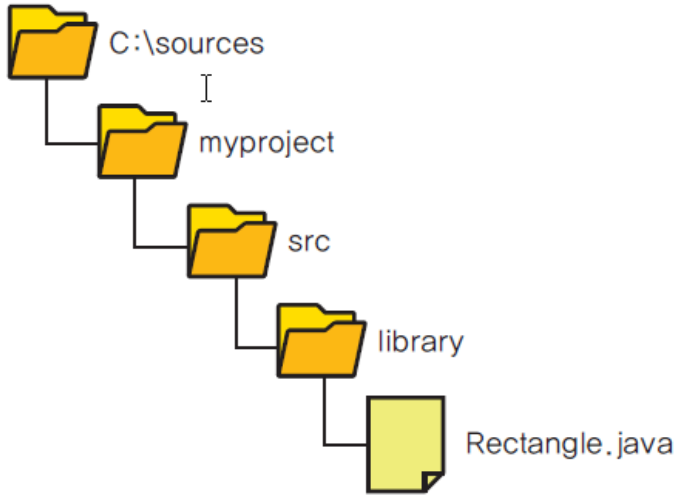
```
import static java.lang.Math.*;
…
double r = cos(PI * theta);        // 클래스 이름 Math 생략
```

소스 파일과 클래스 파일 관리(이클립스)

- 패키지의 계층 구조를 반영한 디렉토리 구조에 소스와 클래스 파일들을 저장한다.
  - 소스 파일

```
package library;
public class Rectangle {
    ...
    ...
}
```

Rectangle.java

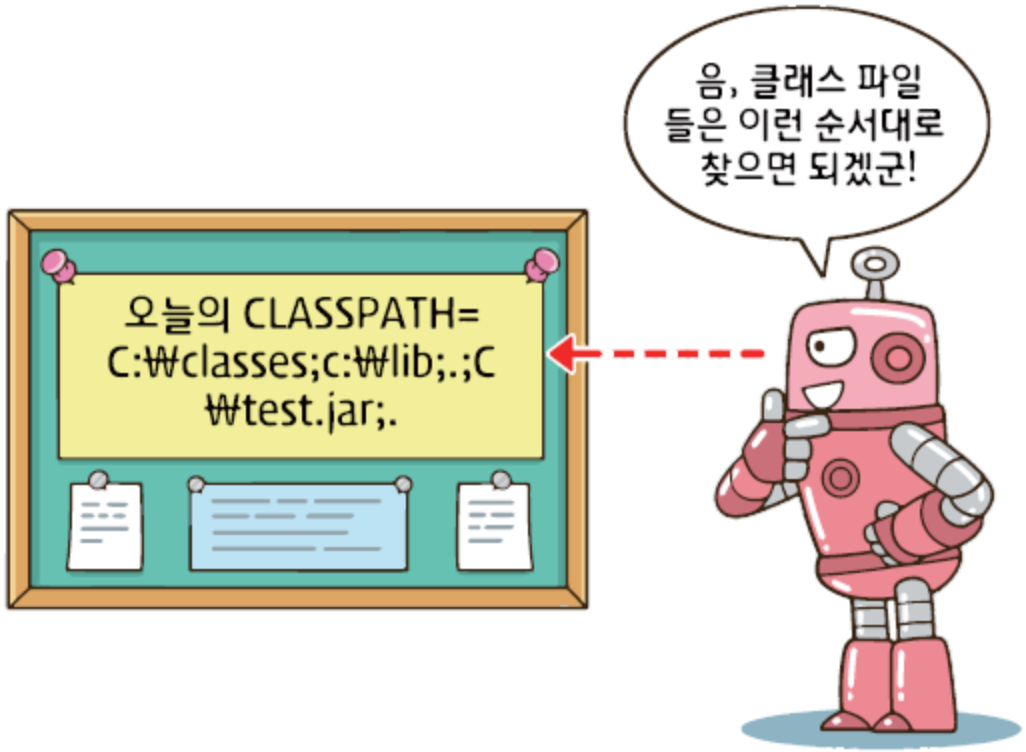


  - 클래스 파일

C:\sources\myproject\bin\library\Rectangle.class

JVM은 어떻게 클래스 파일을 찾을까?

- 자바 가상 머신이 클래스 파일을 찾는 디렉토리들을 클래스 경로(class path)라고 한다.



클래스 경로를 지정하는 방법

- JVM은 항상 현재 작업 디렉토리부터 찾는다.
- 환경 변수인 CLASSPATH에 설정된 디렉토리에서 찾는다. CLASSPATH 지정하는 방법은 다음과 같다.
  - 제어판의 환경변수 설정을 이용하는 방법
  - 명령행에서 CLASSPATH 지정하는 방법

```
C:\>set CLASSPATH=C:\classes;C:\lib;.
```

- JVM을 실행할 때 옵션 -classpath를 사용할 수 있다.

```
C:\>java -classpath C:\classes;C:\lib;. library.Rectangle
```