제 10 장
중첩클래스
Part-2: 익명클래스와 람다식

익명(무명) 클래스

- 익명 클래스(anonymous class)는 클래스 몸체는 정의되지만 이름이 없는 클래스이다
  - 클래스 선언과 인스턴스화가 동시에 일어나며, 이름이 없으므로 재사용이 불가능하다
  - 특정한 형태의 지역클래스(local class)
  - 한 번만 객체를 생성할 필요가 있는 지역 클래스인 경우 지역클래스를 선언하지 않고 익명 클래스를 사용하는 것이 효율적
    - 1회 객체 생성용 지역클래스

익명클래스 선언

- 클래스 선언이 필요없는 일종의 표현식 명령문 형태
  - 세미콜론(;)으로 종료한다
  - 람다 표현식으로 쉽게 변환될 수 있다
  - 생성자 메소드는 선언할 수 없다
- 익명 클래스 구조
  - New 연산자 +
  - 구현할 인터페이스 이름 혹은 상속 받을 수퍼클래스 이름 +
    - 보통 인터페이스를 많이 사용
  - 생성자 인수를 기술하는 "( )" +
    - 인터페이스의 경우에는 생성자가 없으므로 빈 괄호
  - 중괄호를 사용하는 클래스 몸체 +
  - 세미콜론(;)으로 종료

```
new (구현할)인터페이스명(혹은 상속받을 수퍼클래스 명) ( )
{
        //클래스 몸체
};
```

예제

```
interface RemoteControl {
        void turnOn();
        void turnOff();
}
public class AnonymousClassTest {
       public static void main(String args[]) {
              RemoteControl ac = new RemoteControl() {          // 무명 클래스 정의
                     public void turnOn() {
                            System.out.println("TV turnOn()");
                     }
                      public void turnOff() {
                            System.out.println("TV turnOff()");
                     }
              };
              ac.turnOn();
              ac.turnOff();
       }
}
```

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

```
public class HelloWorldAnonymousClass {
    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }
    public void sayHello() {
        // 지역 클래스
        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
            System.out.println("Hello " + name);
            }
        }
        HelloWorld englishGreeting = new EnglishGreeting();
        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Salut " + name);
            }
        };
        HelloWorld spanishGreeting = new HelloWorld() {
            String name = "mundo";
            public void greet() {
                greetSomeone("mundo");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hola, " + name);
            }
        };
        englishGreeting.greet();
        frenchGreeting.greetSomeone("Fred");
        spanishGreeting.greet();

    } // sayHello 메소드 종료

    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp =
                                new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
} // HelloWorldAnonymous 클래스 종료
```

람다식(Lambda Expression)

- 함수 중심의 절차형 프로그래밍 언어는 함수 자체를 포인터 변수 등에 할당 가능 → 함수 포인터
  - 매개변수를 통해 함수를 다른 함수로 전달 가능
- 람다식(lambda expression)은 실행을 목적으로 다른 메소드로 전달될 수 있는 함수 기반의 표현식
- 람다식 : 메소드를 매개변수를 통해 다른 메소드로 전달
  - 함수형 언어의 특성을 도입하여 메소드를 객체처럼 취급할 수 있는 기능
  - 메소드가 필요한 영역에서 즉시로 메소드를 작성하여 사용 가능
  - 일종의 익명 메소드

람다식의 구문

- 람다식은 (arguments list) -> { body } 구문을 사용하여 작성

(매개변수 리스트) → { ... 표현식 혹은 명령문 몸체 ... }

  - 콤마로 구분되는 매개변수들의 리스트는 괄호( ) 안에 기술된다
    - (type1 var1, type2 var2, ...)
    - 매개변수의 데이터 타입을 생략할 수 있다
    - 단지 하나의 매개변수만 있다면 괄호도 생략할 수 있다
  - 하나의 식 혹은 명령문 블록으로 구성되는 몸체 ({ } 사용)
    - a*5 + 6
    - p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25
    - a → **{ return (a*5 + 6) }**
    - p → **{ return** p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25; **}**
    - email → System.out.println(email)

- square_sum(x, y) = (x * x + y * y) can be rewritten in ***anonymous form*** as
  - (x, y) -> (x * x + y * y)

- () -> System.out.println("Hello World");
- (String s) -> { System.out.println(s); }
- () -> 69
- () -> { return 3.141592; };

람다식은 왜 필요한가?

- 메소드를 다른 메소드에 전달 : 프로그램의 단순화를 유도
- 예) 람다식을 사용하여 버튼의 클릭 이벤트를 처리할 수 있다.

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

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

- 함수형 인터페이스(functional interface)
  - 단지 하나의 추상 메소드만을 가지고 있는 인터페이스
  - 함수 인터페이스는 람다식을 사용하면 간단하게 표현 가능

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

// 람다식을 이용한 방법
new Thread( ( ) -> {
        System.out.println("버튼이 클릭되었음!");
}).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
...
```

```
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) {
                        }
            }
  }
}
```

람다식 사용

```
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) {
            }
  }
  }
}
```

함수형 인터페이스와 람다식

- 함수형 인터페이스는 하나의 추상 메소드만 선언된 인터페이스
  - (예) java.lang.Runnable
- 람다식은 함수 인터페이스에 대입할 수 있다.
  - (예) Runnable r = () -> System.out.println("스레드가 실행되고 있습니다.");

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

public class LambdaTest1 {

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

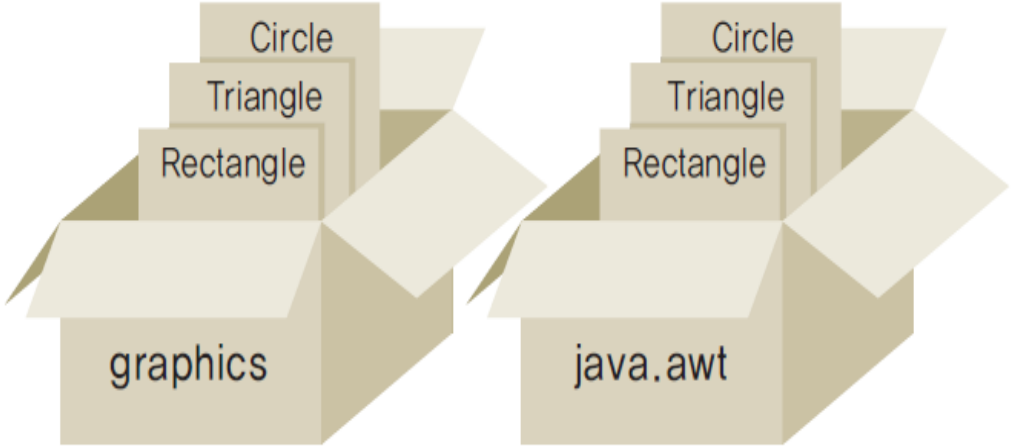
```
Hello Lambda!
```

```
public class Calculator {
    interface IntegerMath {    //함수형 인터페이스
        int operation(int a, int b);
    }
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }
    public static void main(String[] args) {
        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) → a + b;          // 람다식
        IntegerMath subtraction = (a, b) → a – b;      // 람다식
        System.out.println("40 + 2 = " + myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " + myApp.operateBinary(20, 10, subtraction));
    }
}
```

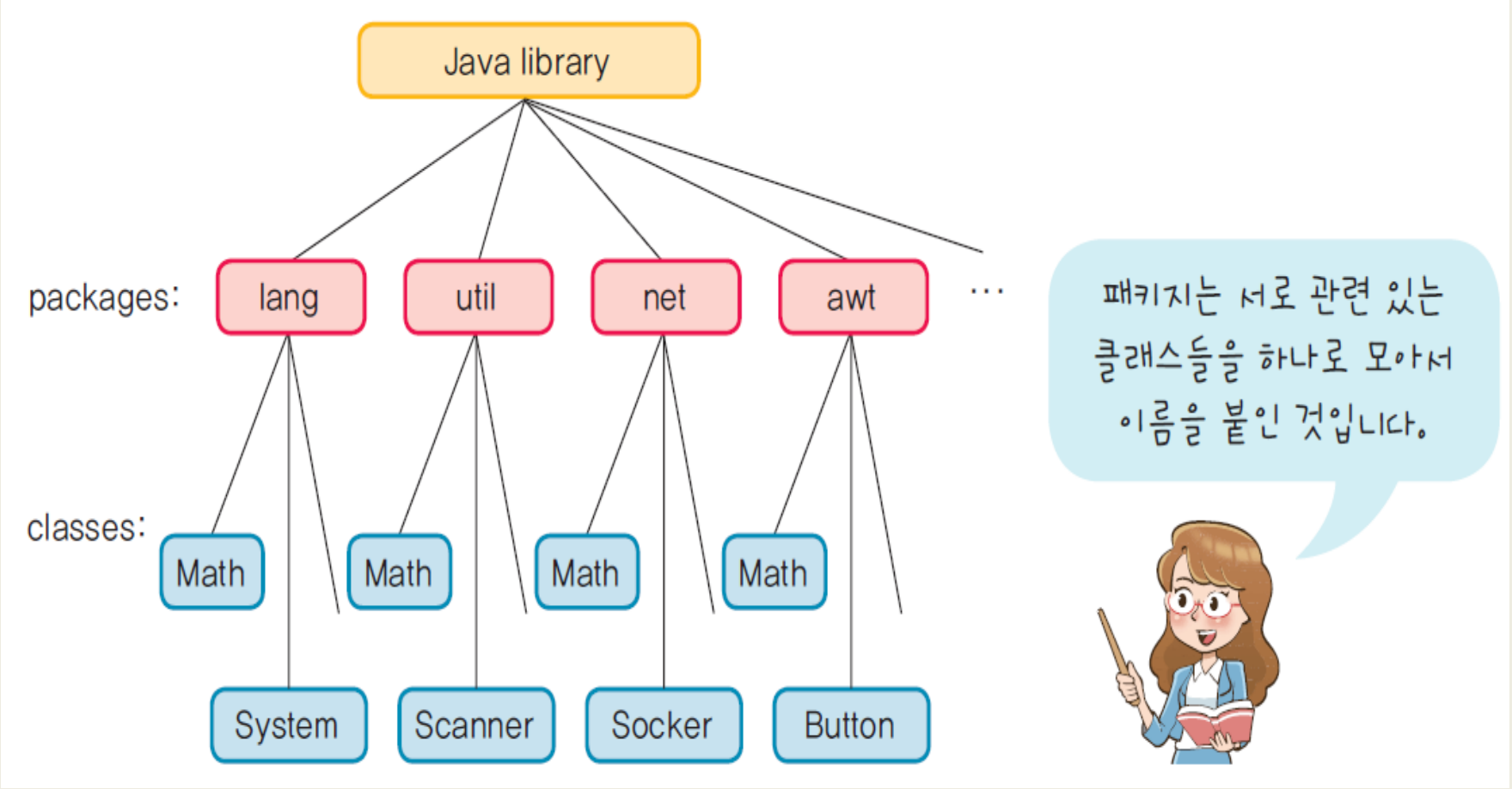
```
40 + 2 = 42
20 - 10 = 10
```

패키지

- 패키지(package)는 서로 관련 있는 클래스나 인터페이스들을 하나로 묶은 것이다.



자바가 제공하는 라이브러리도 기능별로 패키지로 묶여서 제공되고 있다.



패키지를 사용하는 이유

- 패키지를 이용하면 서로 관련된 클래스들을 하나의 단위로 모을 수 있다.
- 패키지를 이용하여서 더욱 세밀한 접근 제어를 구현할 수 있다.
- 패키지를 사용하는 가장 중요한 이유는 바로 “이름공간(name space)” 때문이다.

패키지의 정의

형식

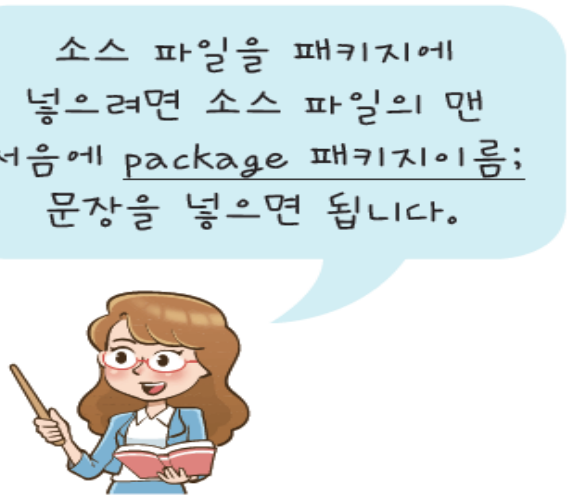
package 패키지이름;

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

Circle.java

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

Rectangle.java



예제

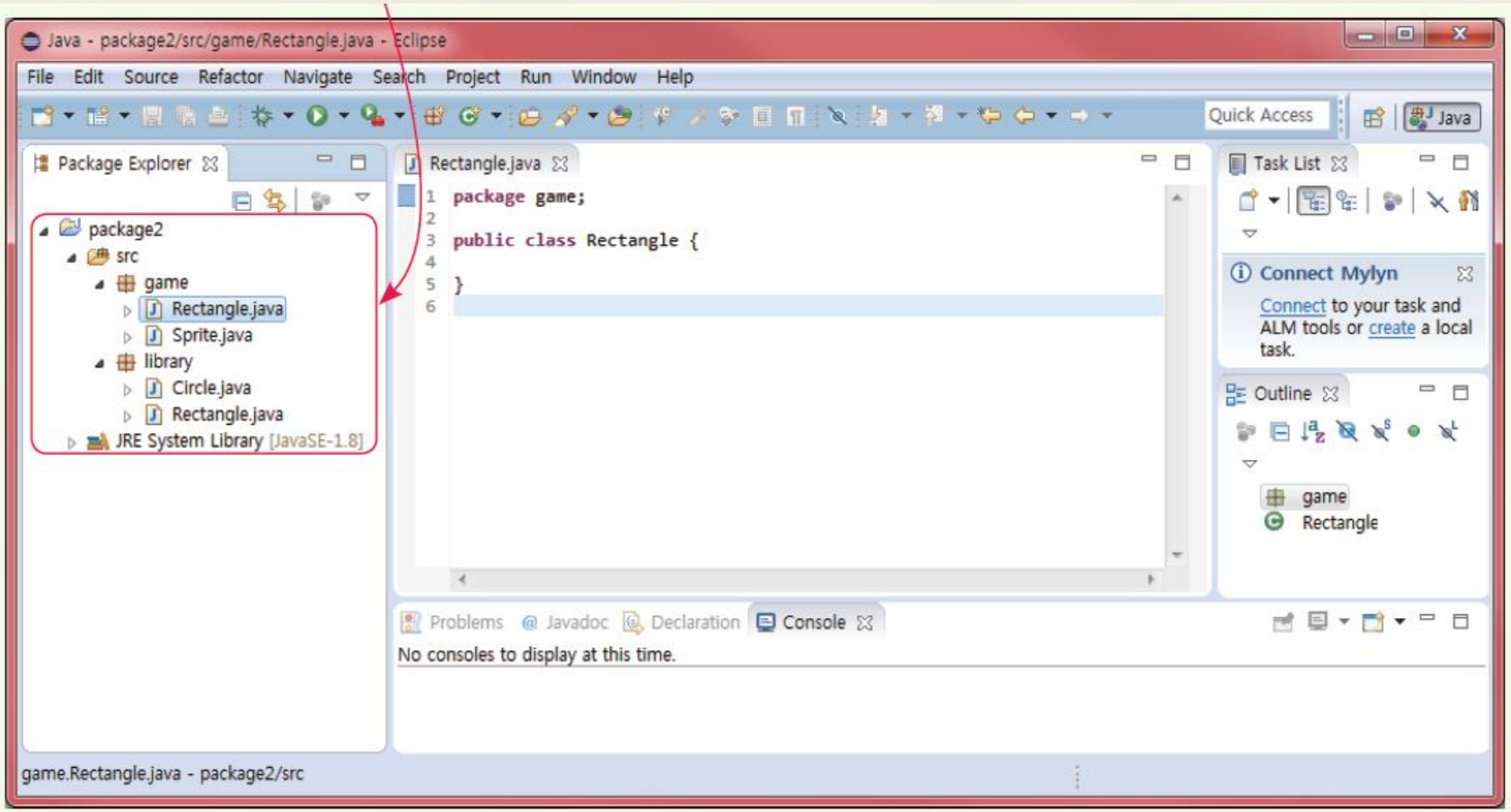
```
package kr.co.company.mylibrary;

public class PackageTest {

  public static void main(String[] args) {
  System.out.println("패키지 테스트입니다.");
  }
}
```

```
D:₩tmp1> javac –d . PackageTest.java
```

패키지 생성하기

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

① 프로젝트 Package2를 생성한다.
② library 패키지를 생성한다.
③ library 패키지에 Rectangle 클래스, Circle 클래스를 추가한다.
④ game 패키지를 생성한다.
⑤ game 패키지에 Rectangle 클래스와 Sprite 클래스를 추가한다


Java - package2/src/game/Rectangle.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
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
Task List
Connect Mylyn
Connect to your task and ALM tools or create a local task.
Outline
game
Rectangle
Problems
Javadoc
Declaration
Console
No consoles to display at this time.
game.Rectangle.java - package2/src

패키지의 사용

- 경로까지 포함하는 완전한 이름으로 참조한다.
  (예) library.Rectangle myRect = new library.Rectangle();

- 원하는 패키지 멤버만을 import한다.
  (예) import library.Rectangle;
  (예) Rectangle myRect = new Rectangle();

- 패키지 전체를 import한다.
  (예) import library.*;

정적 import 문장

- 클래스 안에 정의된 정적 상수나 정적 메소드를 사용하는 경우에 정적 import 문장을 사용하면 클래스 이름을 생략하여도 된다.

(예) import static java.lang.Math.*;

(예) double r = cos(PI * theta);

클래스 경로를 지정하는 3가지의 방법

- 자바 가상 머신은 항상 현재 작업 디렉토리부터 찾는다.
- 환경 변수인 CLASSPATH에 설정된 디렉토리에서 찾는다.

```
D:\tmp1> javac -d . PackageTest.java
```

- 자바 가상 머신을 실행할 때 옵션 -classpath를 사용할 수 있다.

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

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