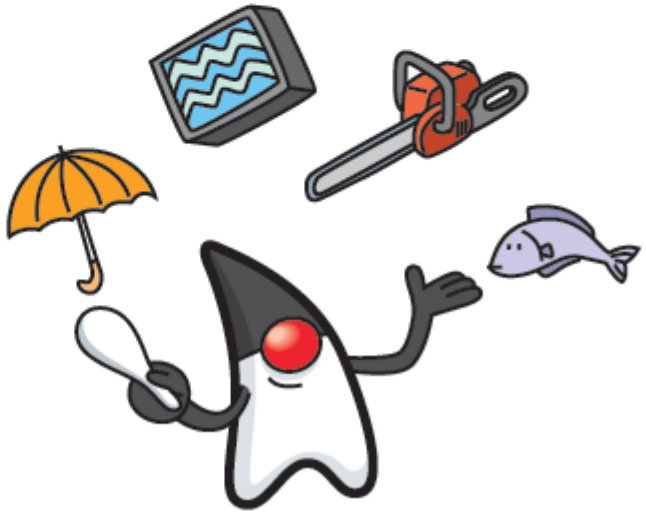
파워자바(개정3판)

7장 추상 클래스, 인터페이스, 중첩 클래스

7장의 목표

1. 인터페이스를 정의하고 사용할 수 있나요?
2. 인터페이스를 이용하여 다중 상속을 구현할 수 있나요?
3. 중첩 클래스의 장점을 이야기할 수 있나요?
4. 클래스 내부에 다른 클래스를 정의하고 사용할 수 있나요?
5. 익명 클래스를 사용하여 코드를 작성할 수 있나요?



추상 클래스

- 추상 클래스(abstract class)는 완전하게 구현되어 있지 않은 메소드를 가지고 있는 클래스를 의미한다.
- 메소드가 미완성되어 있으므로 추상 클래스로는 객체를 생성할 수 없다.
- 추상 클래스는 주로 상속 계층에서 추상적인 개념을 나타내기 위한 용도로 사용된다

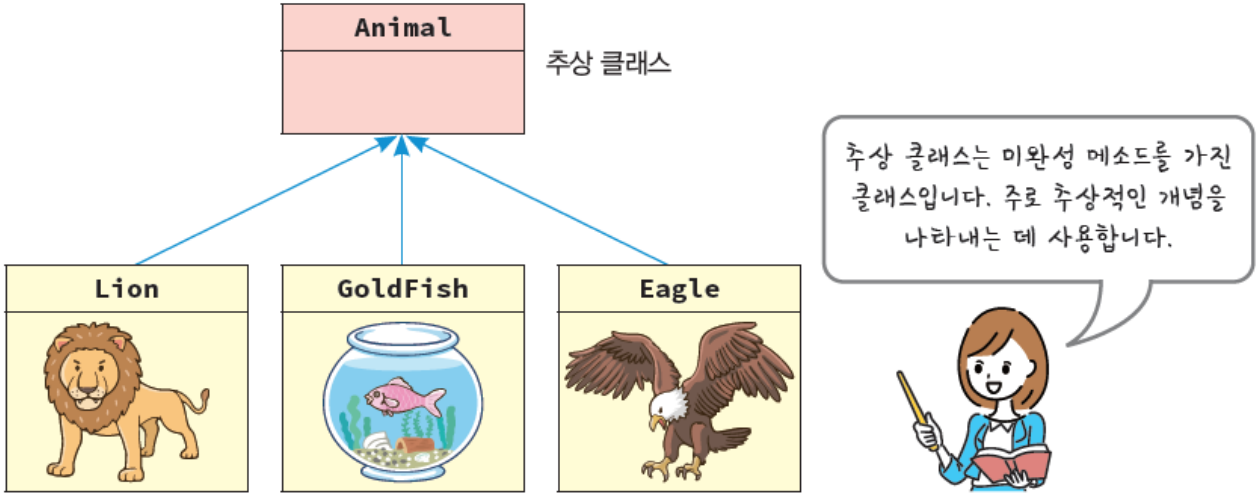


그림 7.1 추상 클래스의 개념

추상 클래스 정의

- 자바에서 추상 클래스를 만들기 위해서는 클래스 선언 시에 앞에 abstract를 붙인다.

```
public abstract class Animal  {
   public abtract void move();
   ...
};
```

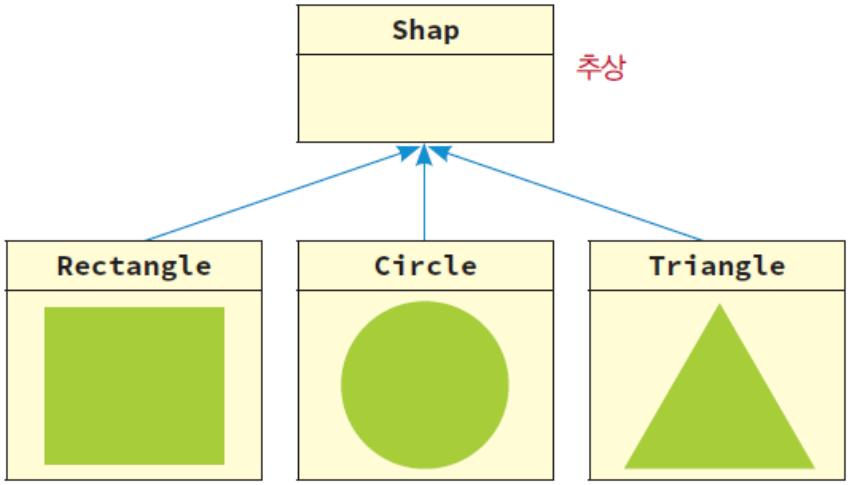
추상 메소드 정의,
;으로 종료됨을 유의!

```
public class Lion extends Animal  {
   public void move() {
      System.out.println("사자의 move() 메소드입니다.");
   }
};
```

추상 클래스를 상속받으면
추상 메소드를 구현하여야 한다.

예제: 도형을 나타내는 클래스 계층 구조

구체적인 예로 도형을 나타내는 클래스 계층 구조를 생각하여 보자. 각 도형은 공통적인 어떤 속성을 가지고 있다. 예를 들면 위치, 회전 각도, 선 색상, 채우는 색 등의 속성은 모든 도형이 공유한다. 또한 도형의 기준점을 이동하는 메소드인 translate()는 모든 도형에서 동일하다. 따라서 이 속성들과 메소드는 추상 클래스인 Shape에 구체적으로 정의된다.



하지만 draw() 메소드를 생각해보자. 도형을 그리는 방법은 각각의 도형에 따라 달라진다. 따라서 draw()의 몸체는 Shape에서는 정의될 수가 없다. 다만 메소드의 이름과 매개 변수는 정의될 수 있다. 이런 경우에 추상 메소드가 사용된다. 즉 draw()는 추상 메소드로 정의되고 draw()의 몸체는 각각의 자식 클래스에서 작성된다.

예제: 도형을 나타내는 클래스 계층 구조

추상 클래스 Shape를 선언한다. 추상 클래스로는 객체를 생성할 수 없다.

AbstractTest.java

```
01  abstract class Shape {
02      int x, y;
03      public void translate(int x, int y) {
04          this.x = x;
05          this.y = y;
06      }
07      public abstract void draw();
08  };
09
10  class Rectangle extends Shape {
11      int width, height;
12      public void draw() {    System.out.println("사각형 그리기 메소드");    }
13  };
14
15  class Circle extends Shape {
16      int radius;
17      public void draw() {    System.out.println("원 그리기 메소드");    }
18  };
```

추상 클래스라고 하더라도 추상 메소드가 아닌 보통의 메소드도 가질 수 있음을 유의하라.

추상 메소드를 선언한다. 추상 메소드를 하나라도 가지면 추상 클래스가 된다. 추상 메소드를 가지고 있는데도 abstract를 class 앞에 붙이지 않으면 컴파일 오류가 발생한다.

자식 클래스 Rectangle에서 부모 클래스의 추상 메소드 draw()가 실제 메소드로 구현한다. 자식 클래스에서 추상 메소드를 구현하지 않으면 컴파일 오류가 발생한다.

예제: 도형을 나타내는 클래스 계층 구조

```
20  public class AbstractTest {
21     public static void main(String args[]) {
22        Shape s1 = new Shape();  // 오류!! 추상 클래스로 객체를 생성할 수는 없다.
23        Shape s2 = new Circle(); // OK!!
24        s2.draw();
25     }
26  };
```

원 그리기 메소드

추상 클래스의 용도

```
abstract class Shape {
   public abstract void draw();
};
```

```
class Shape {
   public void draw() { }
};
```


```
class Circle extends Shape {
   @Override
   public void draw() {
      System.out.println("Circle draw()");
   }
};
```

```
class Circle extends Shape {
   @Override
   public void draw() {
      System.out.println("Circle draw()");
   }
};
```

추상 메소드로 정의되면 자식 클래스에서 반드시 오버라이드하여야 한다. 하지 않으면 오류가 발생한다.

일반 메소드로 정의되면 자식 클래스에서 오버라이드하지 않아도 컴파일러가 체크할 방법이 없다.

중간점검

1. 추상 클래스는 주로 어떤 용도로 사용되는가?
2. 추상 클래스는 추상 메소드가 아닌 일반 메소드를 가질 수 있는가?
3. 추상 클래스만으로 객체를 생성할 수 있는가?

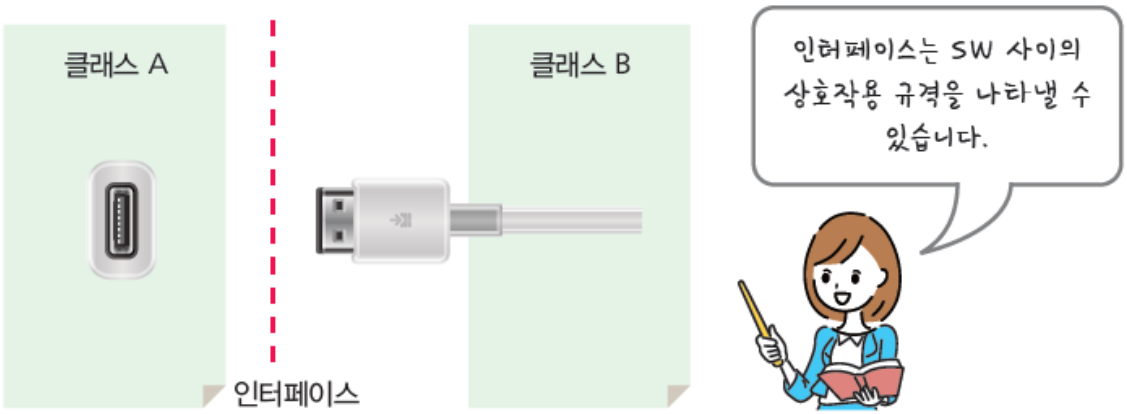
하드웨어 인터페이스

- 컴퓨터 하드웨어에서 인터페이스(interafce)는 서로 다른 장치들이 연결되어서 상호 데이터를 주고받는 규격을 의미한다

USB 인터페이스

인터페이스

- 각자의 클래스를 다른 사람의 클래스와 연결하려면 클래스 간의 상호작용을 기술하는 일종의 규격이 있어야 한다.
- 이러한 규격을 인터페이스(interface)로 정의할 수 있다.

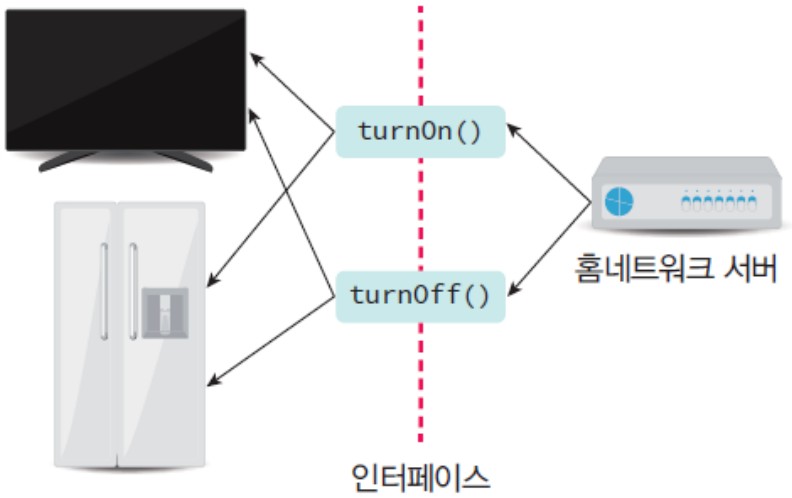


인터페이스의 용도

- 인터페이스는 상속 관계가 아닌, 클래스 간의 유사성을 인코딩하는 데 사용된다.
- 예를 들어, 사람(Human)과 자동차(Car)는 둘 다 달릴 수 있다. 그렇다고 부모 클래스로 Runner를 작성하고, Human과 Car를 Runner 클래스의 자식 클래스로 나타내는 것은 약간 이치에 맞지 않는다.
- 이런 경우에 Runnable 인터페이스를 만들고 이 인터페이스를 양쪽 클래스가 구현하게 하면 된다

인터페이스의 예

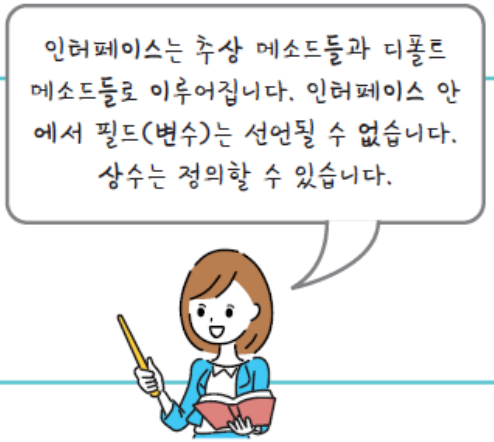
- 예를 들어 홈 네트워크 시스템을 생각하여 보자. 가전 제품과 홈 네트워크 서버 사이에는 가전 제품을 제어할 수 있는 일종의 표준 규격이 필요하다



인터페이스 정의

Syntax: 인터페이스 정의

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




```
public interface RemoteControl {
    // 추상 메소드 정의
    public void turnOn();  // 가전 제품을 켠다.
    public void turnOff(); // 가전 제품을 끈다.
}
```

인터페이스 구현

- 인터페이스는 다른 클래스에 의하여 구현(implement)될 수 있다. 인터페이스를 구현한다는 말은 인터페이스에 정의된 추상 메소드의 몸체를 정의한다는 의미이다.

```
class Television implements RemoteControl {
   boolean on;
   public void turnOn() {
      on = true;
      System.out.println("TV가 켜졌습니다.");
   }
   public void turnOff() {
      on = false;
      System.out.println("TV가 꺼졌습니다.");
   }
}
```

인터페이스 사용

- Television 클래스의 객체를 생성하여 인터페이스에 정의된 메소드를 호출하여 보자.

```
Television t = new Television();
t.turnOn();
t.turnOff();
```

```
Refrigerator r = new Refrigerator();
r.turnOn();
r.turnOff();
```

인터페이스 vs 추상 클래스

- 추상 클래스 사용 권장
  - 만약 관련된 클래스들 사이에서 코드를 공유하고 싶다면 추상 클래스를 사용하는 것이 좋다.
  - 공통적인 필드나 메소드의 수가 많은 경우, 또는 public 이외의 접근 지정자를 사용해야 하는 경우에 추상 클래스를 사용한다.
  - 정적이 아닌 필드나 상수가 아닌 필드를 선언하기를 원할 때 사용한다.

- 인터페이스 사용 권장
  - 관련 없는 클래스들이 동일한 동작을 구현하기를 원할 때 사용한다. 예를 들어서 Comparable과 Cloneable과 같은 인터페이스는 관련없는 클래스들이 구현한다.
  - 특정한 자료형의 동작을 지정하고 싶지만 누가 구현하든지 신경쓸 필요가 없을 때 사용한다.
  - 다중 상속이 필요할 때 사용한다.

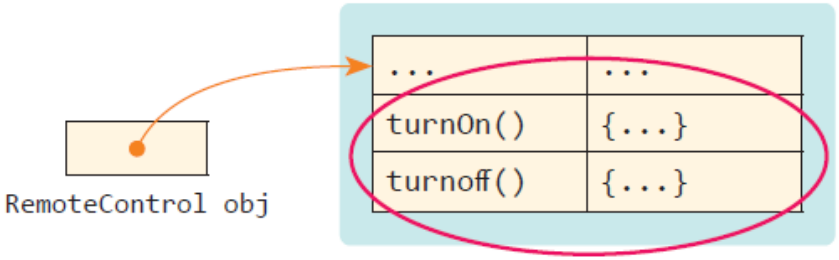
인터페이스와 타입

- 우리가 인터페이스를 정의하는 것은 새로운 자료형을 정의하는 것과 마찬가지이다. 우리는 인터페이스 이름을 자료형처럼 사용할 수 있다.

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

Television 객체이지만 RemoteControl 인터페이스를 구현하기 때문에 RemoteControl 타입의 변수로 가리킬 수 있다.

obj를 통해서는 RemoteControl 인터페이스에 정의된 메소드 만을 호출할 수 있다.



예제: 원격 제어 인터페이스

TestInterface2.java

```
01  interface RemoteControl {
02     void turnOn();
03     void turnOff();
04     public default void printBrand() { System.out.println("Remote Control 가능 TV"); }
05  }
06
07  class Television implements RemoteControl {
08     boolean on;
09     public void turnOn() {
10        on = true;
11        System.out.println("TV가 켜졌습니다.");
12     }
13     public void turnOff() {
14        on = false;
15        System.out.println("TV가 꺼졌습니다.");
16     }
17     @Override
18     public void printBrand() {    System.out.println("Power Java TV입니다.");   }
19  }
```

예제: 원격 제어 인터페이스

```
21  public class TestInterface {
22     public static void main(String args[]){
23        RemoteControl obj = new Television();
24        obj.turnOn();
25        obj.turnOff();
26        obj.printBrand();
27     }
28  }
```

```
TV가 켜졌습니다.
TV가 꺼졌습니다.
Power Java TV입니다.
```

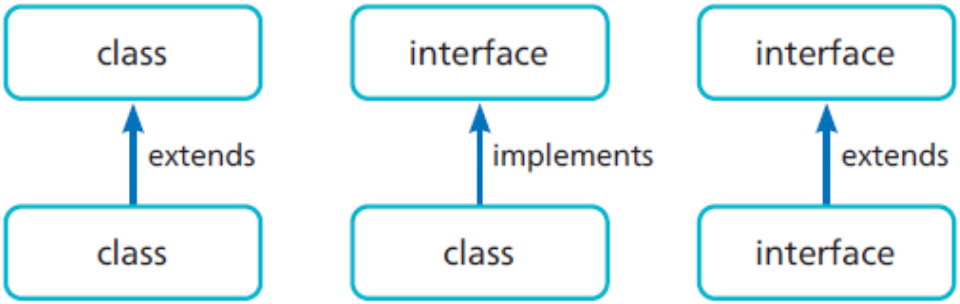
인터페이스도 상속할 수 있다.

- 인터페이스끼리도 상속이 가능하다.

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

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

인터페이스도 다른 인터페이스를 상속받을 수 있다.

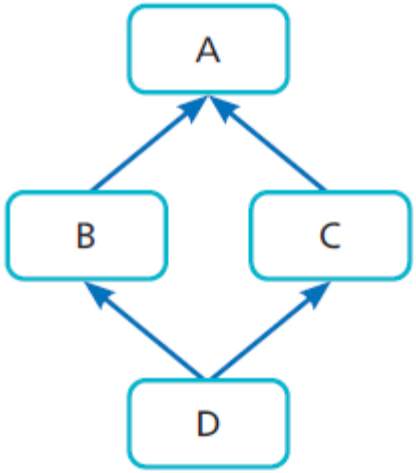


인터페이스를 이용한 다중 상속

- 다중 상속(Multiple inheritance)은 하나의 클래스가 여러 개의 부모 클래스를 가지는 것이다.

다중 상속의 문제점

- 다중 상속은 애매모호한 상황을 만들 수 있기 때문에 자바에서는 금지되어 있다.
- 이것은 흔히 "다이아몬드 문제"로 알려져 있다. 클래스 B와 C가 A로부터 상속을 받는다고 가정하자. 클래스 D는 B와 C에서 상속받는다. 클래스 A에 메소드 myMethod() 가 있는데 이것을 B와 C가 모두 myMethod()를 오버라이드하였다고 가정하자. D를 통하여 myMethod()를 호출하게 되면 어떤 메소드가 호출되는가? B의 메소드인가 아니면 C의 메소드인가?



인터페이스를 이용한 다중 상속

- 동시에 여러 개의 인터페이스를 구현하면 다중 상속의 효과를 낼 수 있다.

FlyingCar1.java

```
01  interface Drivable { void drive(); }
02  interface Flyable { void fly(); }
03
04  public class FlyingCar1 implements Drivable, Flyable {
05     public void drive() { System.out.println("I'm driving"); }
06     public void fly() { System.out.println("I'm flying"); }
07
08     public static void main(String args[]) {
09        FlyingCar1 obj = new FlyingCar1();
10        obj.drive();
11        obj.fly();
12     }
13  }
```

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

인터페이스를 이용한 다중 상속

- 두 번째 방법은 하나의 클래스를 상속받고 또 하나의 인터페이스를 구현하는 것이다

FlyingCar2.java

```
01  interface Flyable { void fly(); }
02
03  class Car {
04     int speed;
05     void setSpeed(int speed){ this.speed = speed; }
06  }
07
08  public class FlyingCar2 extends Car implements Flyable {
09     public void fly() { System.out.println("I'm flying!"); }
10
11     public static void main(String args[]) {
12        FlyingCar2 obj = new FlyingCar2();
13        obj.setSpeed(300);
14        obj.fly();
15     }
16  }
```

```
I'm flying
```

인터페이스에서의 상수 정의

- 인터페이스에서 정의된 변수는 자동적으로 public static final이 되어서 상수가 된다.

```
public interface MyConstants {
    int NORTH = 1;
    int EAST = 2;
    int SOUTH = 3;
    int WEST = 4;
}
```

예제: 다중 상속 예제

- 추상적인 도형을 나타내는 Shape와 그림을 그리는 Drawable 인터페이스를 동시에 상속받아서 Circle 클래스를 정의해보자.

TestInterface2.java

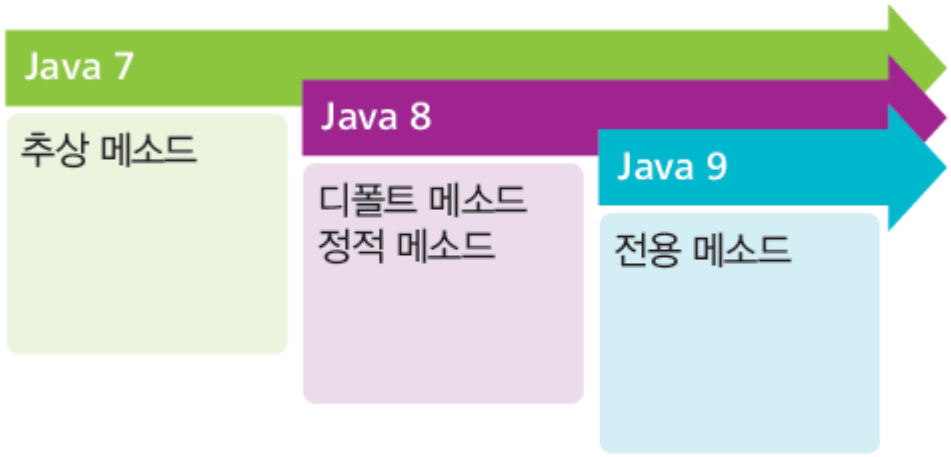
```
01  class Shape {
02     protected int x, y;
03  }
04  interface Drawable{
05    void draw();
06  }
07  class Circle extends Shape implements Drawable {
08     int radius;
09     public void draw() {  System.out.println("Circle Draw at ("+x+", "+y+")");  }
10  }
11  public class TestInterface2 {
12     public static void main(String args[]){
13        Drawable obj = new Circle();
14        obj.draw();
15     }
16  }
```

상속과 동시에 인터페이스를 구현하고 있다.
이것이 가장 일반적인 클래스 정의 형태이다.

```
Circle Draw at (0, 0)
```

디폴트 메소드와 정적 메소드

- Java 8에서 디폴트 메소드와 정적 메소드가 추가되었고 Java 9에서는 전용 메소드(private method)까지 추가되었다.



디폴트 메소드

- 디폴트 메소드(default method)는 인터페이스 개발자가 메소드의 디폴트 구현을 제공할 수 있는 기능이다

DefaultMethodTest.java

```
01  interface MyInterface {
02     public void myMethod1();
03
04     default void myMethod2() {
05        System.out.println("myMethod2()");
06     }
07  }
08
09  public class DefaultMethodTest implements MyInterface {
10     public void myMethod1() {
11        System.out.println("myMethod1()");
12     }
13
14     public static void main(String[] args) {
15        MyClass obj = new MyClass();
16        obj.myMethod1();
17        obj.myMethod2();
18     }
19  }
```

```
myMethod1()
myMethod2()
```

예제: 디폴트 메소드

- 예를 들어서 Drawable 인터페이스에 getSize()라는 메소드가 추가되었다고 하자. 만약 이것을 추상 메소드로만 제공한다면 기존의 코드는 동작하지 않는다. getSize() 메소드를 디폴트 메소드로 정의하여서 기본적인 코드를 붙여준다면 기존의 코드도 변경없이 동작한다.

TestClass.java

```
01  package test;
02  interface Drawable{
03    void draw();
04     public default void getSize(){
05        System.out.println("1024X768 해상도");
06     }
07  }
08  class Circle implements Drawable {
09     int radius;
10     public void draw() {      System.out.println("Circle Draw");    }
11      @Override
12     public void getSize(){    System.out.println("3000X2000 해상도");    }
13
14  public class TestClass  {
```

다시 정의할 수도 있고 아니면 기본 구현을 그대로 사용해도 된다.

예제: 디폴트 메소드

```
15      public static void main (String[] args)    {
16          Drawable d = new Circle();
17          d.getSize();
18          d.draw();
19      }
20  }
```

```
3000X2000 해상도
Circle Draw
```

정적 메소드

- 인터페이스는 전통적으로 추상적인 규격이기 때문에 정적 메소드(static method)가 들어간다는 것은 처음에는 생각할 수도 없었다. Java 8 이전에는 인터페이스에 딸린 정적 메소드를 제공하려면 인터페이스와는 별도의 유틸리티 클래스와 헬퍼 메소드를 생성하여야 했다.

StaticMethodTest.java

```
01  interface MyInterface {
02
03      static void print(String msg) {
04          System.out.println(msg + ": 인터페이스의 정적 메소드 호출");
05      }
06  }
07
08  public class StaticMethodTest {
09
10      public static void main(String[] args) {
11          MyInterface.print("Java 8");
12      }
13  }
```

인터페이스의 정적 메소드이다. JDK8부터 사용이 가능하다.

```
Java 8: 인터페이스의 정적 메소드 호출
```

예제: 정적 메소드

- 하나의 예로 직원을 나타내는 Employee 클래스를 구현할 때 인터페이스 안의 정적 메소드를 사용해보자.

StaticMethodTest2.java

```
01  interface Employable {
02     String getName();
03
04     static boolean isEmpty(String str) {
05        if (str == null || str.trim().length() == 0) {
06           return true;
07        } else {
08           return false;
09        }
10     }
11  }
```

이름을 검사하는 유틸리티 메소드를 만들어서 인터페이스 안에 정적 메소드로 추가할 수 있다.

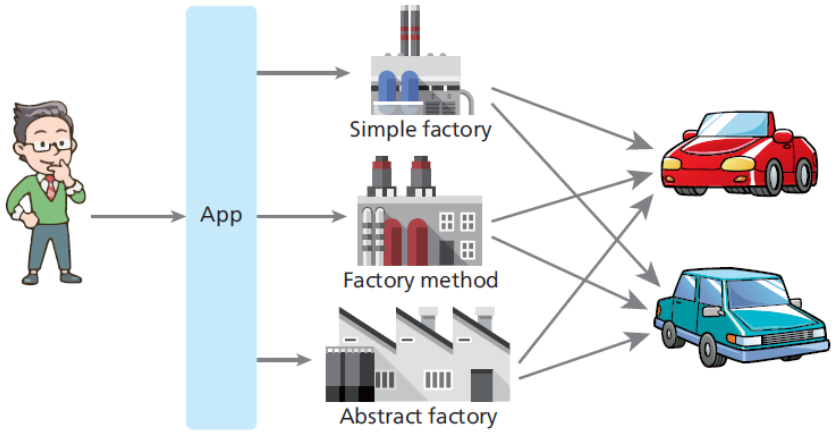
예제: 정적 메소드

```
13  class Employee implements Employable {
14     private String name;
15
16     public Employee(String name) {
17        if (Employable.isEmpty(name) == true)
18           throw new RuntimeException("이름은 반드시 입력하여야 함!");
19        this.name = name;
20     }
21     @Override
22     public String getName() {      return this.name;    }
23  }
24
25  public class StaticMethodTest2 {
26     public static void main(String args[]) {
27        Employable employee1 = new Employee("홍길동");
28        // Employable employee2 = new Employee("");
29     }
30  }
```

정적 메소드를 호출한다.

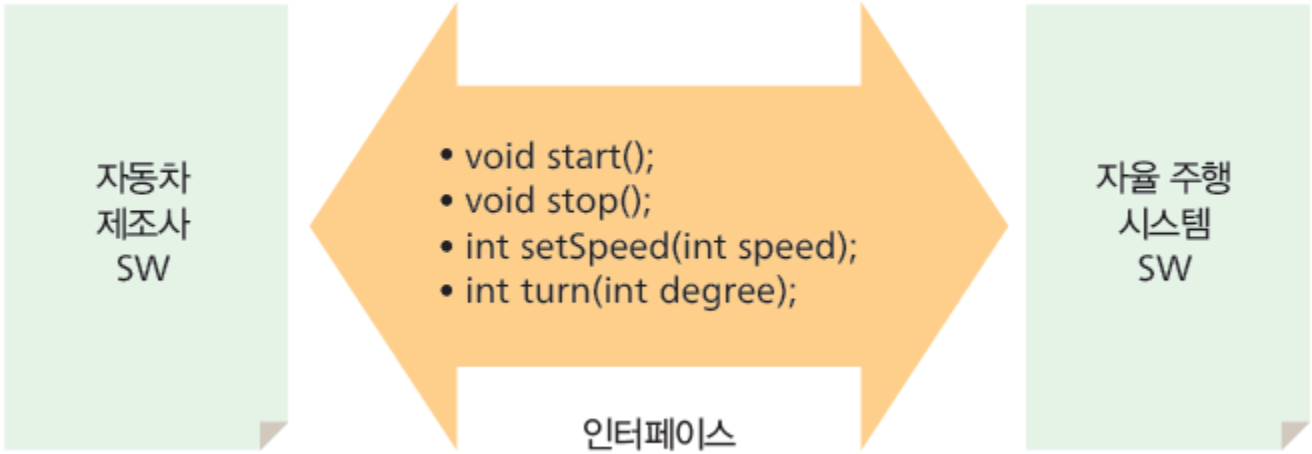
인터페이스와 팩토리 메소드

- 최근에 인터페이스에서도 팩토리 메소드(factory method)가 있는 것이 좋다고 간주되고 있다.
- 팩토리 메소드는 공장처럼 객체를 생성하는 정적 메소드이다. 이것은 디자인 패턴의 하나로서 객체를 만드는 부분을 부모 클래스에 위임하는 패턴이다. 즉 new를 호출하여서 객체를 생성하는 코드를 부모 클래스에 위임한다는 의미이다



Lab: 자율 주행 자동차

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




```
자동차가 출발합니다.
자동차가 속도를 30km/h로 바꿉니다.
자동차가 방향을 15도 만큼 바꿉니다.
자동차가 정지합니다.
```

Sol: 자율 주행 자동차

OperateCar.java

```
01  public interface OperateCar {
02
03     void start();
04     void stop();
05     void setSpeed(int speed);
06     void turn(int degree);
07  }
```

Sol: 자율 주행 자동차

AutoCar.java(자동차 제조사 구현 부분)

```
01  public class AutoCar implements OperateCar {
02     public void start() {
03        System.out.println("자동차가 출발합니다.");
04     }
05
06     public void stop() {
07        System.out.println("자동차가 정지합니다.");
08     }
09
10     public void setSpeed(int speed) {
11        System.out.println("자동차가 속도를 " + speed + "km/h로 바꿉니다.");
12     }
13
14     public void turn(int degree) {
15        System.out.println("자동차가 방향을 " + degree + "도 만큼 바꿉니다.");
16     }
17  }
```

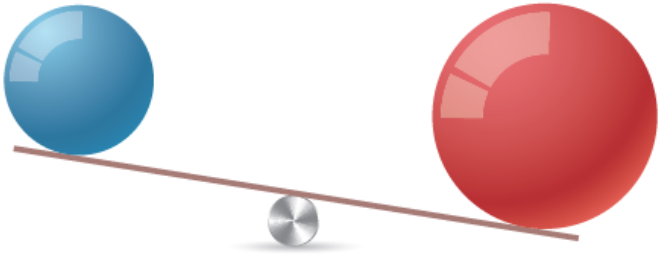
Sol: 자율 주행 자동차

AutoCarTest.java(자율 주행 시스템 업체 부분)

```
01  public class AutoCarTest {
02     public static void main(String[] args) {
03        OperateCar obj = new AutoCar();
04        obj.start();
05        obj.setSpeed(30);
06        obj.turn(15);
07        obj.stop();
08     }
09  }
```

Lab: 객체 비교하기

- Comparable 인터페이스는 객체와 객체의 순서를 비교할 때 사용된다.



```
public interface Comparable {       // 실제로는 제네릭을 사용해서 정의된다.
   int compareTo(Object other);     // -1, 0, 1 반환
}
```

```
Rectangle [width=100, height=30]
Rectangle [width=200, height=10]
Rectangle [width=100, height=30]가 더 큽니다.
```

Sol:

RectangleTest.java

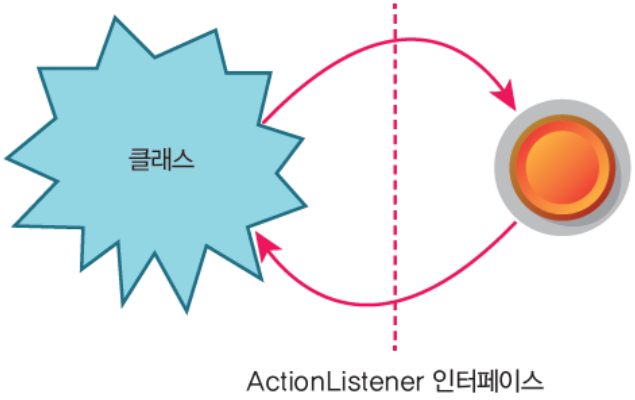
```
01  class Rectangle implements Comparable {
02     public int width = 0;
03     public int height = 0;
04
05     @Override
06     public String toString() {
07        return "Rectangle [width=" + width + ", height=" + height + "]";
08     }
09     public Rectangle(int w, int h) {
10        width = w;
11        height = h;
12        System.out.println(this);
13     }
14     public int getArea() {
15        return width * height;
16     }
17     @Override
18     public int compareTo(Object other) {
19        Rectangle otherRect = (Rectangle) other;
20        if (this.getArea() < otherRect.getArea())
21           return -1;
22        else if (this.getArea() > otherRect.getArea())
23           return 1;
24        else
25           return 0;
26     }
27  }
```

Sol: 객체 비교하기

```
28  public class RectangleTest {
29     public static void main(String[] args) {
30        Rectangle r1 = new Rectangle(100, 30);
31        Rectangle r2 = new Rectangle(200, 10);
32        int result = r1.compareTo(r2);
33        if (result == 1)
34           System.out.println(r1 + "가 더 큽니다.");
35        else if (result == 0)
36           System.out.println("같습니다");
37        else
38           System.out.println(r2 + "가 더 큽니다.");
39     }
40  }
```

Lab: 타이머 이벤트 처리

- 인터페이스가 가장 많이 사용되는 곳은 그래픽 사용자 인터페이스를 구현할 때이다. 예를 들어서 버튼을 눌렀을 때 발생하는 이벤트를 처리하려면 어떤 공통적인 규격이 있어야 한다.

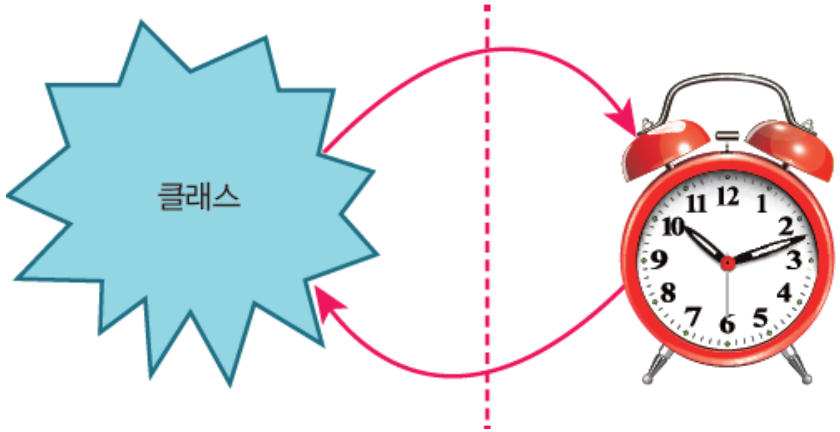



```
public interface ActionListener {
    void actionPerformed(ActionEvent event);
}
```

Lab: 타이머 이벤트 처리

- Timer 클래스는 주어진 시간이 되면 이벤트를 발생시키면서 actionPerformed() 메소드를 호출한다. 이 점을 이용하여서 1초에 한 번씩 다음과 같이 “beep”를 출력하는 프로그램을 작성하여 보자.

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



Sol:

CallbackTest.java

```
01  import java.awt.event.ActionEvent;
02  import java.awt.event.ActionListener;
03  import javax.swing.Timer;
04
05  class MyClass implements ActionListener {
06     public void actionPerformed(ActionEvent event) {
07        System.out.println("beep");
08     }
09  }
10
11  public class CallbackTest {
12     public static void main(String[] args) {
13
14        ActionListener listener = new MyClass();
15        Timer t = new Timer(1000, listener);
16        t.start();
17        for (int i = 0; i < 1000; i++) {
18           try {
19              Thread.sleep(1000);
20           } catch (InterruptedException e) {
21           }
22        }
23     }
24  }
```

ActionListener 인터페이스를 구현한 객체를 생성한다.

Timer에 의하여 1초에 한 번씩 호출된다.

actionPerformed()를 호출해달라고 Timer에 등록한다.

아직 학습하지 않았지만 1초 동안 잤다가, 깨어나는 동작을 1000번 되풀이한다. 1초에 한번씩 호출되는지를 보기 위하여 반복하는 것이다. 단위는 밀리초이다.

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

중간점검

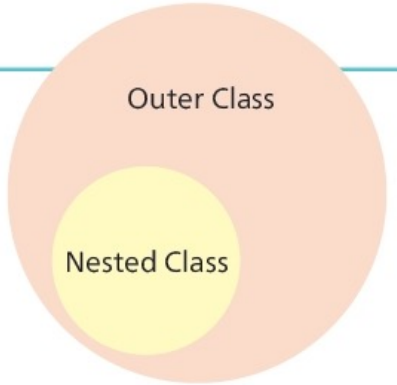
1. 인터페이스로 객체를 생성할 수 있는가?
2. 인터페이스를 정의할 때 사용되는 키워드는?
3. 인터페이스와 클래스의 차이점은 무엇인가?
4. 인터페이스가 가질 수 없는 멤버는 어떤 멤버인가?

중첩 클래스

- 자바에서는 클래스 안에서 클래스를 정의할 수 있다. 내부에 클래스를 가지고 있는 클래스를 외부 클래스(outer class)라고 한다. 클래스 내부에 포함되는 클래스를 중첩 클래스(nested class)라고 한다.

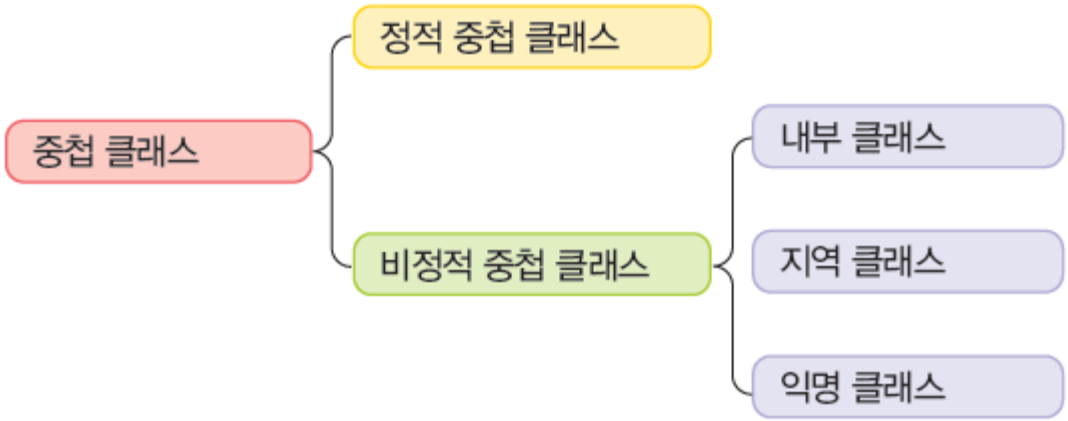
Syntax: 중첩 클래스 정의

```
class OuterClass {
   ...
   class NestedClass {
      ...
   }
}
```



중첩 클래스의 분류

- 정적 중첩 클래스: 앞에 static이 붙어서 내장되는 클래스
- 비정적 중첩 클래스: static이 붙지 않은 일반적인 중첩 클래스
  - 내부 클래스(inner class): 클래스의 멤버처럼 선언되는 중첩 클래스
  - 지역 클래스(local class): 메소드의 몸체 안에서 선언되는 중첩 클래스
  - 익명 클래스(anonymouse class): 수식의 중간에서 선언되고 바로 객체화되는 클래스



내부 클래스

- 클래스 안에 클래스를 선언하는 경우이다.
- 내부 클래스는 외부 클래스의 인스턴스 변수와 메소드를 전부 사용할 수 있다.

InnerClassTest.java

```
01  class OuterClass {
02     private int value = 10;
03
04     class InnerClass {
05        public void myMethod() {
06           System.out.println("외부 클래스의 private 변수 값: " + value);
07        }
08     }
09
10     OuterClass() {
11        InnerClass obj = new InnerClass();
12        obj.myMethod();
13     }
14  }
15
16  public class InnerClassTest {
17     public static void main(String[] args) {
18        OuterClass outer = new OuterClass();
19     }
20  }
```

이것이 바로 내부 클래스이다. 내부 클래스 안에서는 외부 클래스의 private 변수들을 참조할 수 있다.

내부 클래스를 사용한다.

```
외부 클래스의 private 변수 값: 10
```

지역 클래스

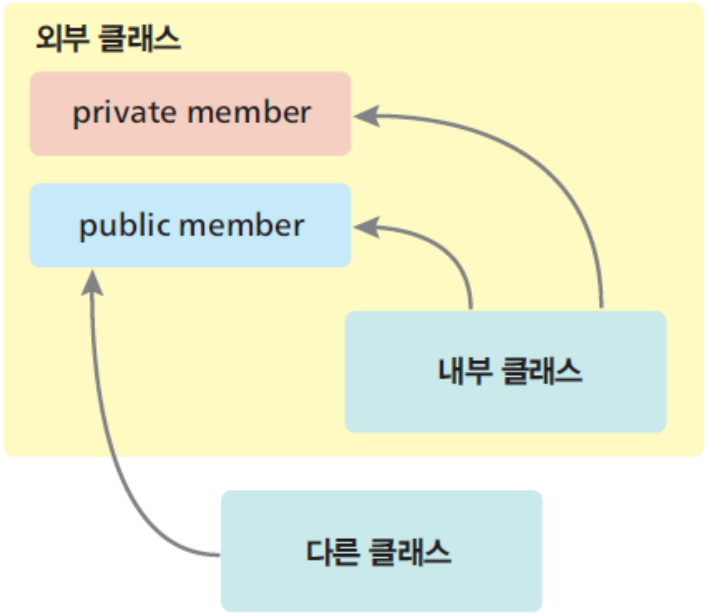
- 지역 클래스(local class)는 메소드 안에 정의되는 클래스이다.

```
01  class localInner {
02     private int data = 30;    // 인스턴스 변수
03
04     void display() {
05        final String msg = "현재의 데이터값은 ";
06
07        class Local {
08           void printMsg() {
09              System.out.println(msg + data);
10           }
11        }
12        Local obj = new Local();
13        obj.printMsg();
14     }
15  }
16
17  public class localInnerTest {
18     public static void main(String args[]) {
19        localInner obj = new localInner();
20        obj.display();
21     }
22  }
```

메소드 display() 안에 클래스 Local이 정의되어 있다. 지역 클래스는 메소드 안에서만 사용이 가능하다. 외부 클래스의 private 변수에 접근할 수 있다.

```
현재의 데이터값은 30
```

중첩 클래스를 사용하는 이유



Lab: 타이머 이벤트 처리

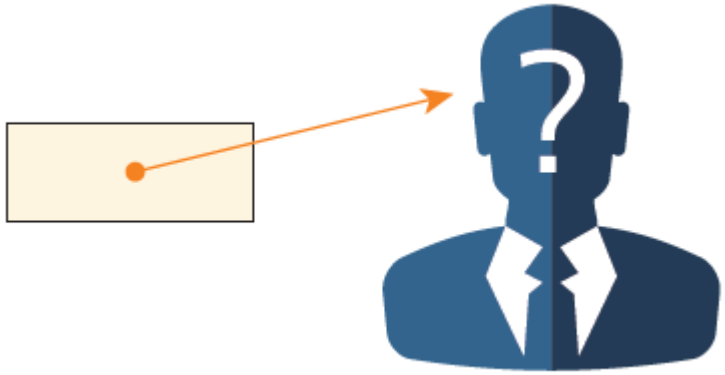
- 타이머 이벤트를 지역 클래스로 다시 작성하여 보자.

```
01  ...
02  public class CallbackTest {
03     public static void main(String[] args) {
04        class MyClass implements ActionListener {
05           public void actionPerformed(ActionEvent event) {
06              System.out.println("beep");
07           }
08        }
09        ActionListener listener = new MyClass();
10        Timer t = new Timer(1000, listener);
11        t.start();
12        for (int i = 0; i < 1000; i++) {
13           try {
14              Thread.sleep(1000);
15           } catch (InterruptedException e) {           }
16        }
17     }
18  }
```

```
beep
beep
beep
```

익명 클래스

- 익명 클래스(anonymous class)는 클래스 몸체는 정의되지만 이름이 없는 클래스이다. 익명 클래스는 클래스를 정의하면서 동시에 객체를 생성하게 된다. 이름이 없기 때문에 한 번만 사용이 가능하다.



익명 클래스 정의

Syntax: 익명 클래스 정의

```
부모클래스 참조변수 = new 부모클래스( ) {
    ...    // 클래스 구현
}
```

상속받고자 하는 부모 클래스의 이름이나 구현하고자 하는 인터페이스의 이름을 적어준다.

```
class Car extends Vehicle {
    ...
}
Car obj = new Car();
```

```
Vehicle obj = new Vehicle() { ....  };
```

익명 클래스의 예

AnonymousClassTest.java

```
01  public interface RemoteControl {
02     void turnOn();
03     void turnOff();
04  }
05
06  public class AnonymousClassTest {
07     public static void main(String args[]) {
08        RemoteControl ac = new RemoteControl() {       // 익명 클래스 정의
09           public void turnOn() {
10              System.out.println("TV turnOn()");
11           }
12           public void turnOff() {
13              System.out.println("TV turnOff()");
14           }
15        };
16        ac.turnOn();
17        ac.turnOff();
18     }
19  }
```

익명 클래스가 정의되면서 동시에 객체도 생성된다.

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

예제: 액션 이벤트 처리

- 앞에서 타이머 이벤트를 내부 클래스로 처리한 적이 있다. 이것을 익명 클래스로 다시 작성하여 보자.

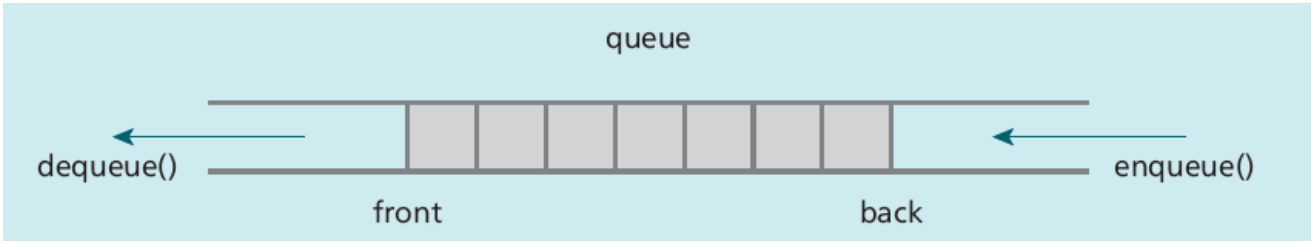
```
01  ...
02  public class CallbackTest {
03     public static void main(String[] args) {
04        ActionListener listener = new ActionListener() {
05           public void actionPerformed(ActionEvent event) {
06              System.out.println("beep");
07           }
08        };
09        Timer t = new Timer(1000, listener);
10        t.start();
11        for (int i = 0; i < 1000; i++) {
12           try {
13              Thread.sleep(1000);
14           } catch (InterruptedException e) {           }
15        }
16     }
17  }
```

익명 클래스가 정의되면서 동시에 객체도 생성된다.

```
beep
beep
beep
```

Mini Project: 큐 구현하기

- 큐(queue)는 요소가 선입선출 방식으로 추가되고 제거되는 자료구조이다.



- 인터페이스 Queue를 설계해보자.
  - q.dequeue() : Queue에서 하나의 항목을 삭제하고 반환한다.
  - q.enqueue(item) : Queue에서 하나의 항목을 추가한다.
  - q.isEmpty() : Queue가 비어 있는지를 검사한다.
- 인터페이스 Queue를 구현하는 클래스 MyQueue 클래스를 작성해보자.

Summary

- 인터페이스는 추상 메소드만을 담고 있는 특수한 클래스이다.
- 인터페이스는 interface 키워드로 정의한다.
- 인터페이스를 상속받는 클래스는 반드시 인터페이스 안에 정의된 추상 메소드들을 구현하여야 한다. 즉 메소드의 몸체를 만들어야 한다.
- 인터페이스를 이용하면 다중 상속의 효과를 낼 수 있다. 인터페이스 안에는 필드가 없기 때문에 다중 상속에서 발생하는 복잡한 문제가 발생하지 않는다.
- 디폴트 메소드(default method)는 인터페이스 개발자가 메소드의 디폴트 구현을 제공할 수있는 기능이다. 디폴트 메소드가 정의되어 있으면 인터페이스를 구현하는 클래스가 메소드의 몸체를 구현하지 않아도 메소드를 호출할 수 있다.
- 중첩 클래스는 클래스 안에 정의되는 클래스를 나타낸다.
- 중첩 클래스에는 내부 클래스, 지역 클래스, 익명 클래스 등이 있다.
- 중첩 클래스를 사용하는 이유는 클래스 안의 멤버들을 자유롭게 접근하기 위해서이다.
- 익명 클래스는 이름이 없이 생성되고 사라지는 클래스이다. 클래스 작성과 객체 생성이 동시에 이루어진다.

Q & A