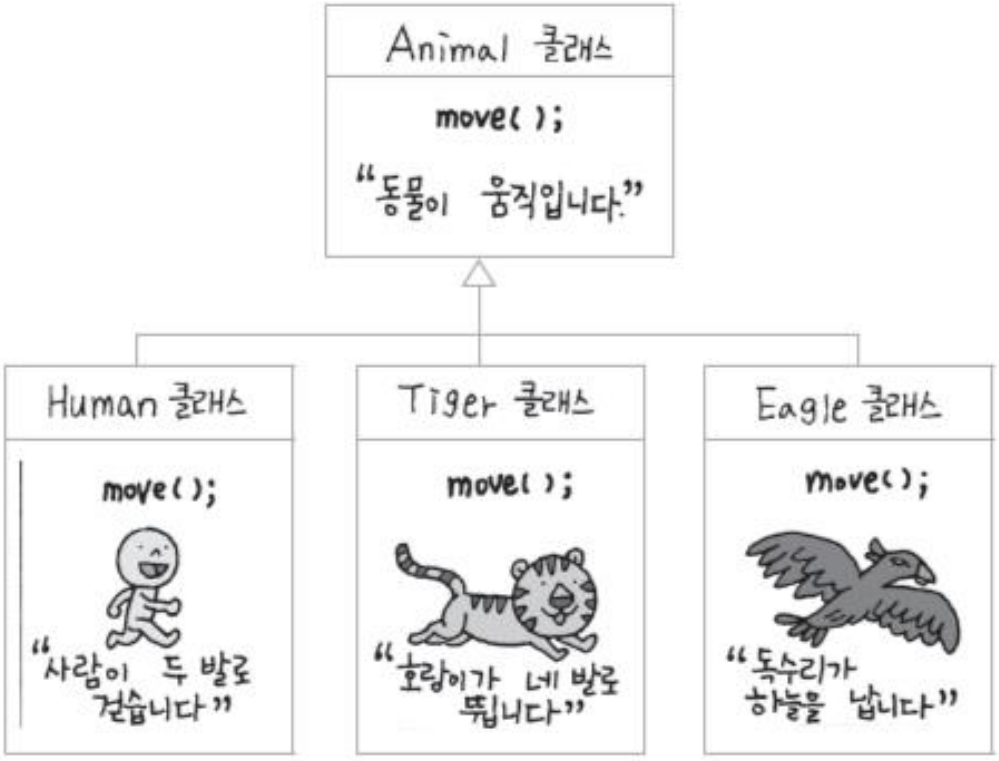
8. 다형성(polymorphism)

다형성

- 객체들의 타입이 다르면 **똑같은 메시지가 전달되더라도 서로 다른 동작을 하는 것.**



상속을 통한 다형성

- 상속은 다형성의 기초로서 사용될 수 있다.

- **메소드 재정의를 통해 다형성을 실현할** 수 있다.
  - **같은 이름의 서로 다른 메서드를 재정의 함**

예: 메소드 재정의를 통한 다형성

```
public class Animal {
    public void move() {
        System.out.println("나는 동물이고 움직인다.");
    }
}
public class Fish extends Animal {
    public void move() {
        System.out.println("나는 수영을 하는 물고기이다.");
    }
}
public class Bird extends Animal {
    public void move() {
        System.out.println("나는 나는 새이다.");
    }
}
```

예: 메소드 재정의를 통한 다형성

```
public class Dog extends Animal {
    public void move() {
        System.out.println("나는 킁킁거리며 돌아다니는 개이다.");
    }
    public void bark() {
        System.out.println("나는 멍멍하고 짓는다.");
    }
}
```

메소드 재정의를 통한 다형성

```
public class Driver {
    public static void main(String [] args) {
        Animal[] animalArray = new Animal[3];
        int index;
        animalArray[0] = new Bird();
        animalArray[1] = new Fish();
        animalArray[2] = new Dog();
     for (index = 0; index < animalArray.length; index++) {
            animalArray[index].move();
        }
    }
}
```

Q: 위 프로그램의 출력 결과는?

다운 캐스팅 – instanceof

하위 클래스가 상위 클래스로 형 변환 되는 것은 묵시적으로 이루어 짐

다시 원래 자료 형인 **하위 클래스로 형 변환 하려면 명시적으로 다운 캐스팅을 해야 함**

=> 이때 원래 **인스턴스의 타입을 체크하는 예약어가 instanceof 임**

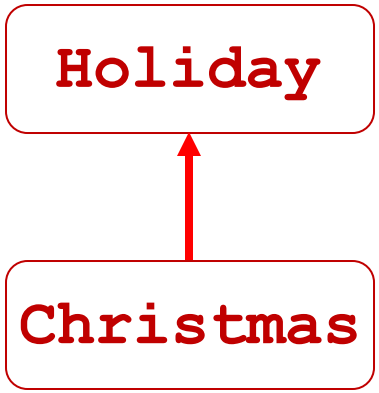
```
public class Driver {
    public static void main(String [] args) {
        Animal[] animalArray = new Animal[3];
        int index;
        animalArray[0] = new Bird();
        animalArray[1] = new Fish();
        animalArray[2] = new Dog();
        for (index = 0; index < animalArray.length; index++) {
            if(animalArray[index] instanceof Bird){
                Bird bird = (Bird) animalArray[index];
                bird.move();
            } else if(animalArray[index] instanceof Fish){
                Fish fish = (Fish) animalArray[index];
                fish.move();
            }else if()..
        }
    }
}
```

예: 상속과 참조

- Holiday 클래스가 Christmas 클래스의 상위 클래스라고 하자. 그리고 Christmas 클래스가 Holiday 클래스의 celebrate 라는 메소드를 재정의 한다고 가정한다.

- 다음 코드에서 **day.celebrate()는 Christmas 클래스에서 정의된 celebrate 메소드를 호출한다.**

```
Holiday day;
day = new Christmas();
day.celebrate();
```



추상 클래스와 인터페이스

추상클래스의 이해

추상적으로 정의할 테니, 사용자가 꼭 재정의(overriding) 하세요.

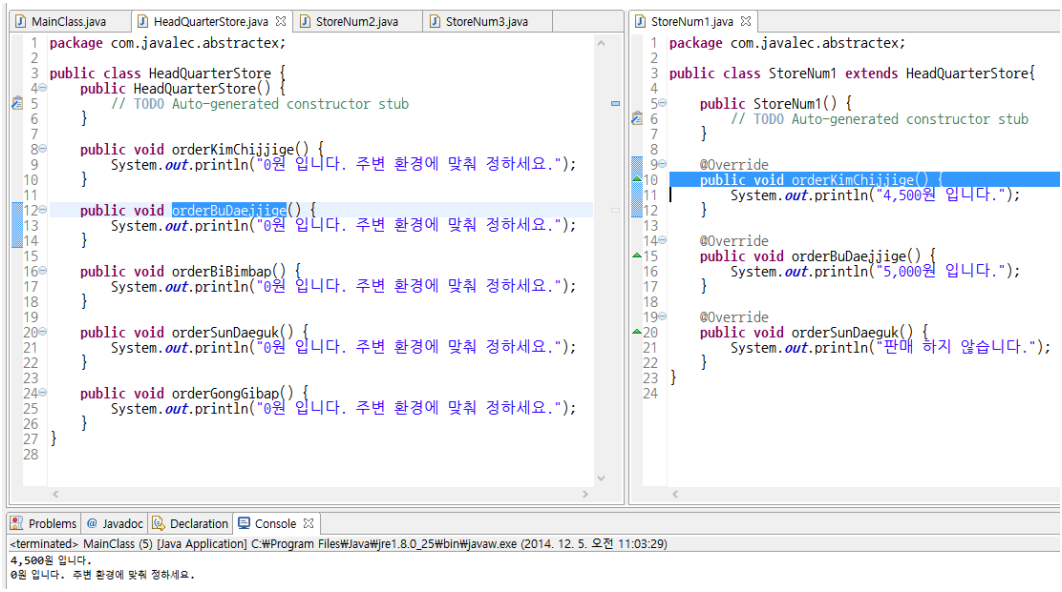
- 우선 본사에서 메뉴에 대한 가격 메소드를 정해 주고, 매장에서는 주변 환경에 맞게 가격을 책정(overriding) 한다.
- 단. **본사에서는 메뉴만 정해 주고, 가격을 매장에 전부 위임 한다.**

본사
김치찌개 – 0원
부대찌게 – 0원
비빔밥 – 0원
순대국 – 0원
공기밥 – 0원

주택가에 매장1호점
김치찌개 – 4,500원
부대찌게 – 5,000원
비빔밥 – 6,000원
순대국 – 판매하지 않음
공기밥 – 1,000원

대학가에 매장2호점
김치찌개 – 5,000원
부대찌게 – 5,000원
비빔밥 – 5,000원
순대국 – 4,000원
공기밥 – 무료

증권가에 매장3호점
김치찌개 – 6,000원
부대찌게 – 7,000원
비빔밥 – 7,000원
순대국 – 6,000원
공기밥 – 1,000원



추상클래스의 이해

추상적으로 정의할 테니, 사용자가 꼭 재정의(overriding) 하세요.

- 비빔밥의 가격이 0원으로 공짜 비빔밥이 되었다.
- 왜 이런 결과가 나온 걸까요? **매장에서는 본사만 믿고 가격을 재정의 하지 않았기 때문임.**
- 그럼 이런 문제를 사전에 예방하려면 어떻게 해야 될까? **본사에서는 '모든 메뉴의 가격을 정하세요.' 라고 매장 점주님께 말해 주면 된다.** 즉 **가격 측정을 강요 하는 행위 이다.**
- => **JAVA프로그램에서도 강제로 부모클래스에서 자식클래스에게 메소드를 강제로 재정의(override)하게 할 수 있다. 그리고 이러한 방법으로 만들어진 클래스를 '추상클래스'라고 한다.**

추상 클래스가 필요한 이유

강제성을 느낄 때 사용 합니다.

HeadQuarterS... | StoreNum2.java | StoreNum3.java

```
1  package com.javalec.abstractex;
2
3  public abstract class HeadQuarterStore {
4      public HeadQuarterStore() {
5          // TODO Auto-generated constructor stub
6      }
7
8      public abstract void orderKimChijjige();
9
10     public abstract void orderBuDaejjige();
11
12     public abstract void orderBiBimbap();
13
14     public abstract void orderSunDaeguk();
15
16     public abstract void orderGongGibap();
17 }
18
```

StoreNum1.java

```
1  package com.javalec.abstractex;
2
3  public class StoreNum1 extends HeadQuarterStore{
4
5      public StoreNum1() {
6          // TODO Auto-generated constructor stub
7      }
8
9      @Override
10     public void orderKimChijjige() {
11         // TODO Auto-generated method stub
12         System.out.println("4,500원 입니다.");
13     }
14
15     @Override
16     public void orderBuDaejjige() {
17         // TODO Auto-generated method stub
18         System.out.println("5,000원 입니다.");
19
20     }
21
22     @Override
23     public void orderBiBimbap() {
24         // TODO Auto-generated method stub
25         System.out.println("6,000원 입니다.");
```

추상 클래스(abstract class)

- **추상 클래스는 하나 이상의 추상 메소드를 포함한다.**
  - 추상 메소드는 **메소드 머리부만 있고 몸체는 없는 메소드이다**
  - 추상 클래스는 보통 메소드들도 포함하고 변수들도 포함할 수 있다.
  - 미완성된 부분들은 **적절한 자식(혹은 자손) 클래스에서 완성된다.**

추상 클래스 선언

- **추상 클래스는 클래스 머리부에 abstract라는 수정자를 포함한다.**
- 하나 이상의 추상 메소드를 포함하는 클래스는 추상 클래스로 선언되어야 한다.
- 추상 메소드도 메소드 머리부에 abstract라는 수정자를 포함해야 한다.

추상 클래스와 메소드

추상 클래스

- 한 개 이상의 추상 메소드를 가지고 있는 클래스
- 객체를 만들 수 없다.
- 이용하려면 하위 클래스를 만들어야 한다.

추상 메소드

- 구현이 안 된 메소드
- 하위 클래스에서 구현이 되어야 한다.

예: public abstract int getArea( );

예: 추상 클래스

```
// 추상 클래스: 일반적인 모양을 모델한다
public abstract class Shape {
    public String name;
    public String getName() {
        return name;
    }
    // 추상 메소드- 몸체가 없다
    public abstract int getArea();
}
```

예: 추상 클래스

```
// 보통 클래스: 직사각형을 모델한다
// 면적을 구할 수 있으므로 메소드 getArea를 구현한다
public class Rectangle extends Shape {
      private int length;
      private int width;

      public Rectangle (String name, int length, int width) {
          this.name = name;
          this.length = length;
          this.width = width;
      }
      public int getArea() {
          return  (length * width);
      }
}
```

다형성을 제한하는 방법

***final* 키워드**: 변수, 메소드와 클래스에 적용된다.

- 변수에 적용되는 경우
  - 값을 상수로 만든다.
  - 값의 변경이 허락되지 않는다.
  - 자바에서 상수를 만드는 방법이다.

- 메소드에 적용되는 경우
  - 메소드를 재정의할 수 없다.
  - 하위 클래스가 메소드의 동작을 변경하지 못하게 하고 싶을 때 사용하라.
  - 불변의 동작(보안 관련, 혹은 "원의 면적 계산")에 대해 사용하라.

- 클래스에 적용되는 경우
  - 하위 클래스를 만들 수 없다.
  - 클래스 전체를 그대로 두고 싶다면 사용하라.

인터페이스(Interface)

- 추상 클래스보다 더 추상적인 클래스
- 다중 상속을 지원하기 위해 만든 특별한 장치
- 상수들과 추상 메소드들의 모음이다.

인터페이스

- 인터페이스의 구문

```
public interface interface_Name {
        public 상수 선언문(들)
        public 메소드 머리부(들)  // 추상 메소드들
}
```

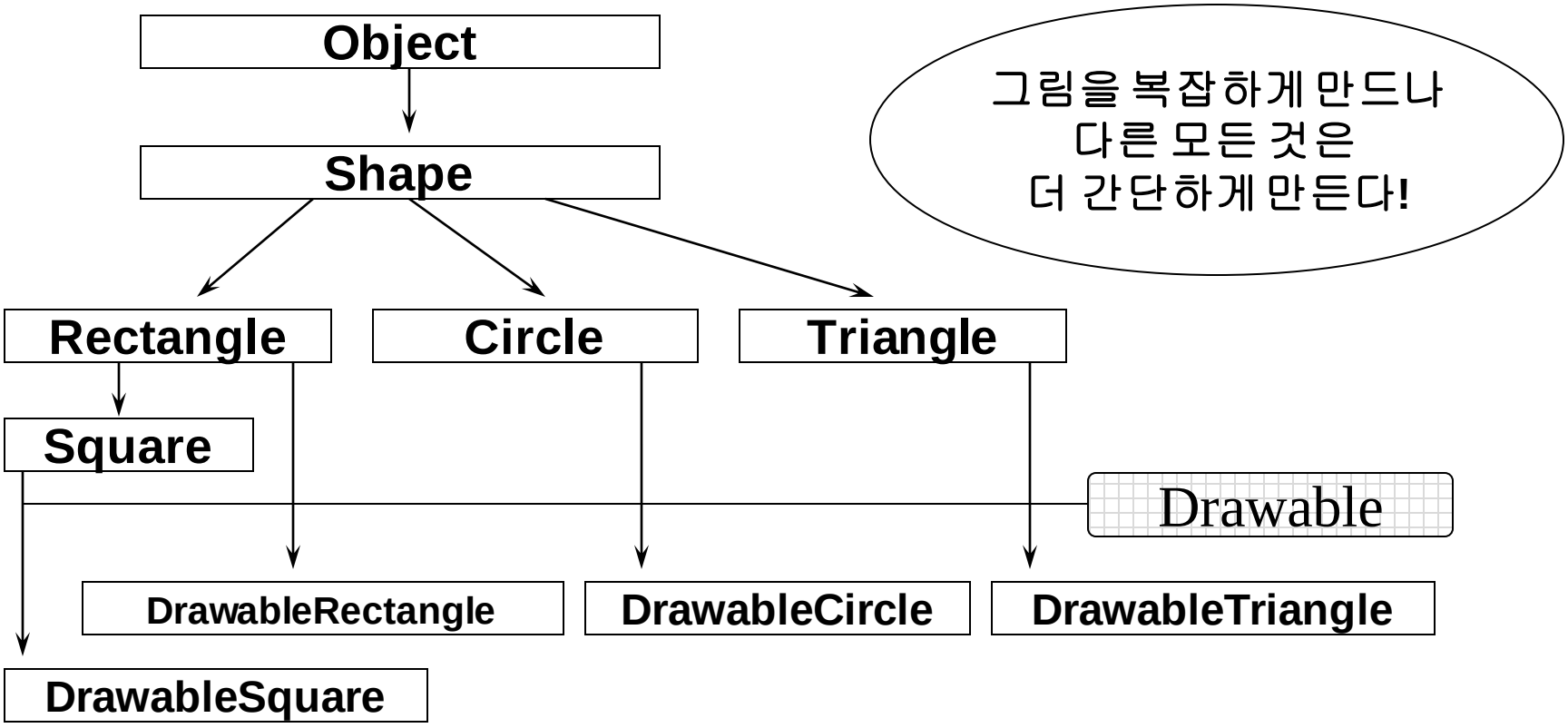
- 한 클래스는 한 인터페이스에 정의된 모든 추상 메소드를 완전하게 구현함으로써 그 인터페이스를 구현한다.

- 한 인터페이스를 구현하는 클래스의 머리부는 다음과 같은 구문을 가진다.

```
class class_Name implements interface_Name
```

예: 인터페이스

아이디어: **Shape** 클래스 계층 구조를 그대로 두고 기능을 확장하도록 하는 인터페이스를 만든다...



예: 인터페이스

기존의 Shape 계층 구조를 그대로 두고 기능을 추가한다.
예를 들면, 기존의 Rectangle을 그릴 수 있게(drawable) 만든다.

```
public interface Drawable {
        public void drawMe(Graphics g);
}
public class DrawableRectangle extends Rectangle implements Drawable
   {
   public void drawMe (Graphics g) {
       g.drawRect(0, 0, Length, Width);
    }
}
```

인터페이스를 통한 다형성

- 인터페이스의 이름이 한 참조 변수의 데이터 형으로 사용될 수 있다.

- 한 인터페이스를 참조하는 변수는 그 인터페이스를 구현하는 어떤 클래스의 어느 객체를 참조하기 위해 사용될 수 있다.

예: 인터페이스를 통한 다형성

```
public interface Printable {
    public void printMe( );
}

public class PrintableText extends TextFile implements Printable {
    // Printable 인터페이스를 구현하기 위해 printMe( ) 메소드
    public void printMe( ) {
        // 텍스트 파일을 출력하는 코드
    }
}

public class PrintableGraphics extends Graphics implements Printable {
    // Printable 인터페이스를 구현하기 위해 printMe( ) 메소드
    public void printMe( ) {
        // Graphics 객체를 출력하는 코드
    }
}
```

예: 인터페이스를 통한 다형성

- 인터페이스의 융통성은 다형적 참조들을 만드는 것을 가능하게 한다.
- 같은 인터페이스를 구현하는 여러 객체들 사이에서 비슷한 다형적 참조를 만들 수 있다.
- 예:

```
Printable printer;

printer = new PrintableText();

printer.printMe();

printer = new PrintableGraphics();

printer.printMe();
```

설명: printer라는 참조 변수는 처음에는 PrintableText 객체를 가리키고 나중에는 PrintableGraphics 객체를 가리킨다.

내부 클래스

- 한 클래스는 변수들과 메소드들 외에 다른 클래스(들)을 포함할 수 있다.
- 한 클래스 내에 선언된 또 다른 클래스는 내부 클래스라고 부른다.
- 내부 클래스를 둘러싸고 있는 클래스는 외부 클래스라고 부른다.
- 내부 클래스는 외부 클래스의 변수들과 메소드들을 사용할 수 있다.

내부 클래스

- 내부 클래스는 외부 클래스의 각 객체와 연관된다.
- 내부 클래스는 두 개의 클래스들 사이에 밀접한 관계가 있고 어떤 다른 클래스에 의해 접근되지 않는 경우에만 만든다.
- 내부 클래스는 별도의 바이트코드 파일을 만든다.

예제 프로그램 작성

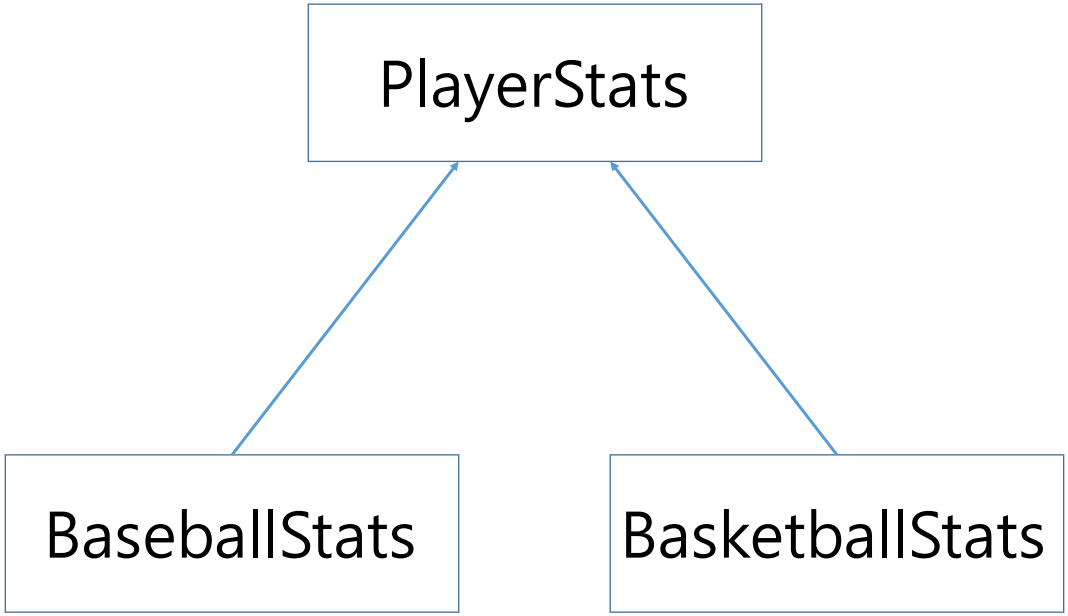
문제: 야구와 농구 선수의 통계를 기록하는 간단한 프로그램을 작성하라.

- 각 선수는 공통적으로 이름, (소속) 팀명과 득점수를 가진다.
- 야구 선수는 추가적으로 안타수와 에러수를 가진다.
- 농구 선수는 추가적으로 도움수와 리바운드수를 가진다.
- 선수의 이름과 소속팀명을 넘겨 받아 객체를 생성할 수 있어야 한다.
- 객체를 생성할 때 각종 기록(득점수, 안타수, 에러수, 도움수 혹은 리바운드수)은 0으로 초기화해야 한다.
- 선수의 각종 기록을 각각 알려 줄 수 있어야 한다.
- 선수의 모든 데이터를 한꺼번에 알려 줄 수 있어야 한다.
- 선수의 각종 기록을 적절하게 변경하거나 주어진 값을 이용하여 새로 계산할 수 있어야 한다. 예를 들면, 야구 선수가 안타를 치면 안타수는 1만큼 증가하고 농구 선수가 리바운드를 하면 리바운드수가 1만큼 증가한다. 농구 선수가 2점 슛을 성공시키면 득점수는 2만큼 증가하나 3점 슛을 성공시키면 득점수는 3만큼 증가한다. 반면에 야구 선수가 홈에 들어 오면 득점수는 1만큼 증가한다.

필요한 클래스들

- PlayerStats 클래스: 야구 선수와 농구 선수에 공통적인 데이터와 연산을 나타낸다
- BaseballStats 클래스: 야구 선수에만 해당되는 데이터와 연산을 나타낸다
- BasketballStats 클래스: 농구 선수에만 해당되는 데이터와 연산을 나타낸다
- Driver 클래스: 세 개의 클래스들을 시험한다

클래스 계층도



PlayerStats 클래스 설계

- 변수들
  - player: 선수의 이름
  - team: 선수의 팀명
  - score: 선수의 득점수

- 메소드들
  - 생성자 메소드
  - getScore: 득점수를 반환한다
  - toString: 객체의 모든 데이터를 반환한다
  - earnScore(추상 메소드): 득점수를 주어진 값만큼 증가시킨다

★ 클래스 구현: 교재의 프로그램 11.6

BaseballStats 클래스 설계

- 변수들
  - hits: 안타수
  - errors: 에러수

- 메소드들
  - 생성자 메소드
  - earnScore: 득점수를 주어진 값만큼 증가시킨다
  - gainHit: 안타수를 1만큼 증가시킨다
  - commitError: 에러수를 1만큼 증가시킨다
  - getHits: 안타수를 반환한다
  - getErrors: 에러수를 반환한다
  - toString: 객체의 모든 데이터를 반환한다

★ 클래스 구현: 교재의 프로그램 11.7

BaseballStats 클래스 설계

- 변수들
  - assists: 도움수
  - rebounds: 리바운드수

- 메소드들
  - 생성자 메소드
  - earnScore: 득점수를 주어진 값만큼 증가시킨다
  - gainAssists: 도움수를 1만큼 증가시킨다
  - gainRebounds: 리바운드수를 1만큼 증가시킨다
  - getAssists: 도움수를 반환한다
  - getRebounds: 리바운드수를 반환한다
  - toString: 객체의 모든 데이터를 반환한다

★ 클래스 구현: 교재의 프로그램 11.8

요약

- 추상 클래스
- 추상 메소드
- 인터페이스
- 다형성
- 내부 클래스
- 예제 프로그램 작성