* 今天学习目标

 * 能够掌握面向对象的接口

 * 能够了解枚举的构建和使用

 * 能够掌握内部类的构建和使用

 * 能够掌握Lambda表达式的使用

 * 能够掌握类与类之间的的关系

---

 * 能够掌握面向对象的接口

        * 生活中接口：扩展，规范（协议）

        * 接口语法：

            * Interface,jdk1.8之前（静态常量，抽象方法，省略public static final，public abstract，jdk1.8 默认方法，静态方法，jdk9 私有方法）

        * USB案例，计算机案例

        * 接口与抽象区别

        * JDK1.8前后接口一些变化

 * 能够了解枚举的构建和使用

        * enum

        * public static final int SEASON_SPRING=0;

        * public static final int SEASON_SUMMER=1;

        * enum SEASON{

            SPRING,SUMMER

            }

         * 更加直观，类型安全

 * 能够掌握内部类的构建和使用

       * 内部：成员内部类，局部内部类，静态内部类

                匿名内部（接口，父类存在）

                * 按钮点击案例

* 能够掌握Lambda表达式的使用

    * JDK1.8 出来，语法糖

    * 简化内部类编写

    * 接口，只用一个抽象方法

    * () ->{};

    * ::

* 能够掌握类与类之间的的关系

    * 泛化（Generalization）,实现(Realization)，组合(Composition)，聚合(Aggregation)，关联(Association)，依赖(D...)

---

* 回顾

    * 抽象类

        * 抽象方法 （ abstract ）没有方法体

        * 能不能构造器：可以，能不能变量：可以

        * ...

    * 类的加载过程和初始化

        * 类的加载时机

            * 类加载：延迟加载

            * 第一次使用类的信息

            * new，继承，JVM标明启动类（文件名与类同名），静态变量或者静态方法，假如有final静态变量，就不加载，Class.forName("类全名称")

        * 类的加载过程

            * 加载（class文件-->内存，java.lang.Class）-->连接（验证，准备（给静态变量分配内存，初始化静态变量），解析（符号引用替换直接引用（常量池））） -->初始化（静态-->父类静态代码块-->子类静态代码块-->父类-->非静态代码块-->构造器-->子类-->非静态代码块-->构造器）-->使用-->卸载（gc）

        * 类加载器:

            * 根类加载器(%JAVA_HOME%/jre/lib) BootStrap ClassLoader

            * 扩展类加载器(%JAVA_HOME%/jre/lib/ext) Extension ClassLoader

* 系统类加载器(classpath) Application ClassLoader

* 自动类加载器

* 双亲委派机制（坑爹机制）

* private，public ,protected,default

* 多态（100--->1）

* 父类类型 父类变量=new 子类对象（）；

* 继承，重写父类方法，向上转型

* instanceOf

* 能够掌握面向对象的接口

* 生活中的接口

* 花洒（假如花洒坏了，怎么办？？？）

* 计算机中的接口有哪些?

* USB接口, 耳机接口, 网线接口, 电源接口...,计算机中的接口可以扩展计算机的功能, 连接更多的外设, 有该接口就扩展了这个功能, 没有这个接口就不具备这个功能

不同厂家生产的USB设备,插到电脑上都能使用, 是因为他们都遵循相同的协议(标准)

* 接口概述

* Java中的接口的作用是用来扩展类的功能; 也可以理解为一组操作规范(协议)

* Java接口是Java语言中一种引用类型，是方法的集合，接口的内部主要就是封装了

方法，包含抽象方法（JDK 7及以前），默认方法和静态方法（JDK 8），JDK 9 私有方法。

* 定义接口

```
[修饰符] interface 接口名 {
 接口中的方法默认使用public abstract修饰
 接口中的字段默认使用public static final修饰
 在接口中还可以定义static修饰的方法，default修饰的方法，private修饰的方法
}
```

* 使用接口

```
[修饰符]  class  类名  implements   接口名{
    需要重写接口中的抽象方法
}
```

* 接口特性

```
* 接口是一种引用数据类型，接口不能实例化对象
* 在JDK8中，接口中包括抽象方法（JDK 7及以前），
  静态常量（JDK 7及以前） ，静态方法（JDK8），默认方法（JDK8）
* 接口支持多继承
* 一个类可以实现多个接口，需要重写所有接口的所有抽象方法，否则需要定义为抽象类
```

* 案例USB接口案例

```
public interface USB {
//  public static final int LENGTH=12;
    int LENGTH=12;// 可以省略public static final，它是常量，不是变量，演示语法无意义

//  public abstract void service();
     // USB 的服务
     void service();//  可以省略public abstract
```

```
}

public class UDiskUSBImpl implements USB{

    @Override
    public void service() {
        System.out.println("UDISK 实现USB 通信协议");
    }

}
public static void main(String[] args) {
        UDiskUSBImpl uDisk=new UDiskUSBImpl();
        uDisk.service();
  }
```

* 业务案例：计算机类，实现计算机类使用USB鼠标、USB键盘

    * 计算机类，包含运行功能、关机功能、使用USB设备功能

    * USB接口，包含开启功能、关闭功能

    * 鼠标类要实现USB接口，具备点击的方法

    * 键盘类，要实现USB接口，具备敲击的方法

```
public interface USB {
    void open();//开启功能
    void close();//关闭功能
}
public class Mouse implements USB{

    @Override
    public void open() {
        System.out.println("鼠标开启，红灯闪呀闪呀");
    }

    @Override
    public void close() {
        System.out.println("鼠标关闭，红灯熄灭");
```

```
    }

    public void click() {
        System.out.println("鼠标单击");
    }

}
public class KeyBoard implements USB{

    @Override
    public void open() {
        System.out.println("键盘开启，绿灯闪呀闪呀");
    }

    @Override
    public void close() {
        System.out.println("鼠标关闭，绿灯熄灭");
    }

    public void keyBoarding() {
        System.out.println("键盘打字");
    }

}
public class Computer {
    // 计算机开启运行功能
    public void run() {
        System.out.println("计算机运行");
    }

    // 使用USB设备功能
    public void useUSB(USB usb) {
        if(usb!=null) {
            usb.open();
            if(usb instanceof Mouse) {
                ((Mouse)usb).click();
            }else if(usb instanceof KeyBoard) {
                ((KeyBoard)usb).keyBoarding();
            }
            usb.close();
```

```
        }
    }
    //计算机关闭
    public void shutDown() {
        System.out.println("计算机关闭");
    }
}
public class Client {
    public static void main(String[] args) {
        Computer computer=new Computer();
        //开机运行
        computer.run();
        // 计算机使用鼠标
        Mouse mouse=new Mouse();
        computer.useUSB(mouse);
        // 计算机使用键盘
        KeyBoard keyBoard=new KeyBoard();
        computer.useUSB(keyBoard);
        //关机
        computer.shutDown();
    }
}
* 运行结果
    计算机运行
    鼠标开启，红灯闪呀闪呀
    鼠标单击
    鼠标关闭，红灯熄灭
    键盘开启，绿灯闪呀闪呀
    键盘打字
    鼠标关闭，绿灯熄灭
    计算机关闭
```

* 接口与抽象类异同点
    * 相同点
        * 都可以定义抽象方法
        * 都不能实例化对象
        * 都可以定义静态常量和静态方法

* 抽象方法都需要被重写

* 不同点

* 意义不同：抽象类是对对象的抽象，接口是功能的封装

* 定义方式不同：abstract class ，interface

* 内容不同

* 抽象类除了抽象方法外,还有构造方法等普通类有的所有成员

* 接口中除了抽象方法外, 还可以使用default修饰某个方法, 注意接口中没有构造方法,

接口在JDK8中只有四部分内容: 抽象方法, 静态常量, 静态方法, 默认方法

* 使用方式不同：抽象类需要被子类extends继承，接口需要被实现类implements实现

* 抽象类只支持单继承, 接口支持多继承

* 使用场景不同

* 仅仅对类进行功能的扩展,优先选择接口

* 除了功能外,还需要保存数据,只能使用抽象类

* JDK1.8 接口

* Java8之前，Java接口纯粹是契约的集合。

* Java8之前，接口不能升级。因为在接口中添加一个方法，会导致老版本接口的所有实现类需要需要重新实现该方法

* Java8之后，lambda表达式作为核心出现，为了配合lambda表达式，于是引入了默认方法和静态方法

* 默认方法理论上抹杀了Java接口与抽象类的本质区别-前者是行为契约的集合，后者是接口与实现的结合体。

```java
public interface MInteferface {
    int AGE=12; // public static final int AGE=12; 常量

    void say(); // public abstract void say(); 抽象方法

    // JDK1.8 提供默认方法
    default void sayHello() {
        System.out.println("say helloWorld");
    }

    public static void speak() {
        System.out.println("speak helloWorlod");
    }
}
```

* 能够了解枚举的构建和使用

  * 枚举概述

    * 枚举（Enum）的全称为 Enumeration， 是 JDK 1.5 中引入的新特性

    * 枚举（Enum）是一个特定类型的类，枚举都是java.lang.Enum的隐式子类。

  * 枚举好处

    * 比起使用常量，枚举类更加直观，类型安全

  * 语法格式

    * [访问权限] enum 枚举名｛ 枚举值列表 ｝

  * 案例

```
* 静态常量
public class Season {
    public static final int SEASON_SPRING = 1;
    public static final int SEASON_SUMMER = 2;
    public static final int SEASON_FALL = 3;
    public static final int SEASON_WINTER = 4;

}
public class SeasonTest {

    public void switchSeason(int season) {
        switch (season) {
        case Season.SEASON_SPRING:
```

```
            System.out.println("春天适合踏青...");
            break;
        case Season.SEASON_SUMMER:
            System.out.println("夏天适合游泳...");
            break;
        case Season.SEASON_FALL:
            System.out.println("秋天适合旅游...");
            break;
        case Season.SEASON_WINTER:
            System.out.println("冬天适合滑雪...");
            break;

        default:
            break;
        }
    }
 }

public static void main(String[] args) {
        SeasonTest seasonTest=new SeasonTest();
        seasonTest.switchSeason(Season.SEASON_SPRING);
}
结果:
春天适合踏青...

枚举形式
public enum SeasonEnum {
    SPRING,SUMMER,FALL,WINTER
}
public class SeasonTest {

    public void switchSeason(SeasonEnum season) {
        switch (season) {
        case SPRING:
            System.out.println("春天适合踏青...");
            break;
        case SUMMER:
            System.out.println("夏天适合游泳...");
            break;
        case FALL:
```

```
            System.out.println("秋天适合旅游...");
            break;
        case WINTER:
            System.out.println("冬天适合滑雪...");
            break;

        default:
            break;
        }
    }
}

public static void main(String[] args) {
        SeasonTest seasonTest=new SeasonTest();
        seasonTest.switchSeason(SeasonEnum.WINTER);
}

结果：冬天适合滑雪...
```

* 能够掌握内部类的构建和使用

* 概述：将一个类定义在另一个类里面或者一个方法里面，这样的类称为内部类

* 内部类分类：成员内部类, 静态内部类(了解), 局部内部类(了解), 匿名内部类（掌握）

* 成员内部类 ：定义在类中方法外的类。

* 定义格式： class 外部类 { class 内部类{ } }

* 作用：成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）

* 应用：一个事物内部还包含其他事物，就可以使用内部类这种结构，例如汽车（Car）和引擎（Engine）

```
public class A implements IGood,IGood2{
    class C{

    }
    @Override
    public void showGood() {
        class D{

        }
    }

    @Override
    public void showInfo() {
//      IGood.super.showInfo();
        IGood2.super.showInfo();
```

成员内部类

```
public class A implements IGood,IGood2{
    private String name;
    class C{
        public void test() {
            System.out.println(name);
        }
    }
```

成员内部的好处：可以无条件访问外部类 private 属性

```
public class A implements IGood,IGood2{
    private String name;
    class C{
        String name;
        public void test() {
            System.out.println(name);
            System.out.println(A.this.name);
        }
    }
```

当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。

```
public static void main(String[] args) {
    A a=new A();
    A.C c=a.new C();
}
```

产生外部类的对象

* 静态内部类就是使用static修饰的成员内部类, 一般在静态方法中创建静态内部类对象

* 局部内部类是在方法体中定义的内部类

* 匿名内部类是内部类的简化写法。它的本质是一个带具体实现的父类或者父接口的 匿名的子类对象。 开发中，最常用到的内部类就是匿名内部类了。

* 点击事件案例

```
没使用匿名内部
public interface OnClickListener {
    void onClick();
}

public class Button {
    private Scanner input=new Scanner(System.in);
    private boolean click=false;
    public void setOnClickListener(OnClickListener listener) {
        System.out.println("请点击（点击（true）/false）");
        if (listener != null) {
            click=input.nextBoolean();
            if (click) {
                listener.onClick();
                click = false;
            }
        }
    }
}
public class MOnClickListener implements OnClickListener{

    @Override
    public void onClick() {
        System.out.println("哥们，我被点了...");
    }

}
public static void main(String[] args) throws InterruptedException {
        Button btn=new Button();
        MOnClickListener mOnClickListener=new MOnClickListener();
        btn.setOnClickListener(mOnClickListener);
}
```

```
结果：
请点击（点击（true）/false）
true
哥们，我被点了...

使用匿名内部类
public static void main(String[] args) {
        Button btn=new Button();
        btn.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick() {
                System.out.println("姐们，你被点了...");
            }
        });
        btn.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick() {
                System.out.println("你为什么要点我呢..");

            }
        });
    }

结果：
请点击（点击（true）/false）
true
姐们，你被点了...
请点击（点击（true）/false）
true
你为什么要点我呢..
```

* 能够掌握Lambda表达式的使用

* Lambda表达式（JDK8）主要是替换了原有匿名内部类的写法，也就是简化了匿名内部类的写法

```
public static void main(String[] args) {
        Button btn=new Button();
        btn.setOnClickListener(()->{
            System.out.println("姐们，你被点了...");
        });
        btn.setOnClickListener(()->{
            System.out.println("你为什么要点我呢..");
        });
}
结果:
请点击（点击（true）/false）
true
姐们，你被点了...
请点击（点击（true）/false）
true
你为什么要点我呢..

```

* lambda语法结构如下：

    * (parameters) ->{ statements; }

        * 小括号内的语法与传统方法参数列表一致：无参数则留空；多个参数则用逗号分隔。

        * -> 是新引入的语法格式，代表指向动作。

        * 大括号内的语法与传统方法体要求基本一致。

    * (parameters) -> expression

        * 省略写法的规则为：

            * 小括号内参数的类型可以省略

            * 如果小括号内有且仅有一个参数，则小括号可以省略

            * 如果大括号内有且仅有一个语句，则无论是否有返回值，都可以省略大括号、return关键字及语句分号。

* Lambda表达式语法示例：

    * 不需要参数,返回值为 8 :() -> 8

* 接收一个参数(数字类型),返回其3倍的值： x -> 5 * x

* 接受2个参数(数字),并返回他们的差值： (x, y) -> y – x

* 接收2个int型整数,返回他们的和： (int x, int y) -> x + y

```
* 不需要参数,返回值为 8  :() -> 8
public interface AgeInterFace {
    int getAge();
}
public class UseAge {
    public void useAge(AgeInterFace ageInterFace) {
        System.out.println(ageInterFace.getAge());
    }
}
public static void main(String[] args) {
        UseAge u=new UseAge();
//      u.useAge(new AgeInterFace() {
//
//          @Override
//          public int getAge() {
//              return 8;
//          }
//      });
        u.useAge(()->8);
    }
结果:
8

* 接收一个参数(数字类型),返回其3倍的值:   x -> 5 * x
public interface AgeInterFace {
    int getAge(int age);
}

public class UseAge {
    public void useAge(AgeInterFace ageInterFace,int age) {
        System.out.println(ageInterFace.getAge(age));
    }
}
public static void main(String[] args) {
        UseAge u=new UseAge();
```

```
        /*u.useAge(new AgeInterFace() {

            @Override
            public int getAge(int age) {
                return 3 * age;
            }
        }, 5);*/
        u.useAge(age->age*3, 5);
    }
 结果：15

 接受2个参数(数字),并返回他们的差值：   (x, y) -> y - x
 public interface AgeInterFace {
    int getAge(int oldAge,int newAge);
}
public class UseAge {
    public void useAge(AgeInterFace ageInterFace,int oldAge,int newAge) {
        System.out.println(ageInterFace.getAge(oldAge,newAge));
    }
}
public static void main(String[] args) {
        UseAge u=new UseAge();
        /*u.useAge(new AgeInterFace() {

            @Override
            public int getAge(int oldAge, int newAge) {
                // TODO Auto-generated method stub
                return newAge-oldAge;
            }
        }, 8, 18);*/
        u.useAge((x,y)->y-x, 8, 18);
       // u.useAge((int x,int y)->y-x, 8, 18);
    }
结果:10

```

* Lambda表达式对返回的实例类型有比较严格的要求：

    * 必须是接口

* 接口中只有一个需要实现的抽象方法，因此如果接口中有超过1个抽象方法需要实现的情况并不适用于lambda表达式

* 每一个lambda表达式都对应一个接口类型。

* “函数式接口”是指仅仅只包含一个抽象方法的接口，每一个该类型的lambda表达式都会被匹配到这个抽象方法。

* 默认方法不算抽象方法，所以也可以给函数式接口添加默认方法

* 为了适配lambda表达式

  * 接口中只有一个需要实现的抽象方法,可以使用 @FunctionalInterface

```
@FunctionalInterface
interface IGood {
        void display();
        void say();
    }
}
```

不能写两个抽象方法

* JDK8中引入了一个新的操作符(::)

  * :: 操作符主要用作静态方法、成员方法或构造方法的绑定

```
public class FunBindOperation {

    public static void main(String[] args) {
        ICalucator c=Math::min;//判断min方法
        int min=c.min(1, 3);
        System.out.println(min);
    }

}
@FunctionalInterface
interface ICalucator{
    int min(int a,int b);
}
```

绑定静态方法

```java
package com.lg.lambda;

public class FunBindOperation {

    public static void main(String[] args) {
        ICalucator c=Math::min;//判断min方法
        int min=c.min(1, 3);
        System.out.println(min);
    }

}
@FunctionalInterface
interface ICalucator{
    int min(int a,int b);
}

```

```java
public class FunBindOperation2 {
    public void display() {
        System.out.println("good display");
    }

    public static void main(String[] args) {
        FunBindOperation2 o2=new FunBindOperation2();
        IGood gd=o2::display;
        gd.display();
    }
@FunctionalInterface
interface IGood {
        void display();
//      void say();
    }
}
```

绑定成员方法

Problems Javadoc Declaration Search Console

```
<terminated> FunBindOperation2 [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (2018年8月19日 下午4:30:43)
good display
```

```java
package com.lg.lambda;

public class FunBindOperation2 {
    public void display() {
        System.out.println("good display");
    }

    public static void main(String[] args) {
        FunBindOperation2 o2=new FunBindOperation2();
        IGood gd=o2::display;
        gd.display();
    }
 @FunctionalInterface
 interface IGood {
        void display();
//      void say();
    }
}

```

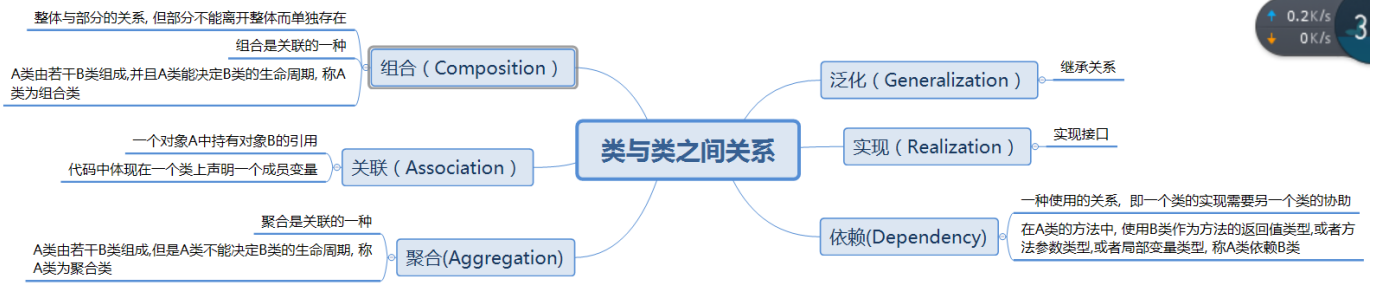
```java
public class FunBindOperation3 {
    public void display() {
        System.out.println("good display");
    }

    public static void main(String[] args) {
        IGood good=Good::new;
        Good g=good.createGood("java in thinking", 20);
        System.out.println(g.name+":"+g.price);
    }

}
@FunctionalInterface
interface IGood {
        Good createGood(String name,double price);
}
class Good {
    String name;
    double price;
    public Good(String name, double price) {
        super();
        this.name = name;
```

运用在构造器上

```java
package com.lg.lambda;

public class FunBindOperation3 {
    public void display() {
        System.out.println("good display");
    }

    public static void main(String[] args) {
        IGood good=Good::new;
        Good g=good.createGood("java in thinking", 20);
        System.out.println(g.name+":"+g.price);
    }

}
@FunctionalInterface
interface IGood {
      Good createGood(String name,double price);
}
class Good {
    String name;
    double price;
    public Good(String name, double price) {
        super();
        this.name = name;
        this.price = price;
    }

}

```

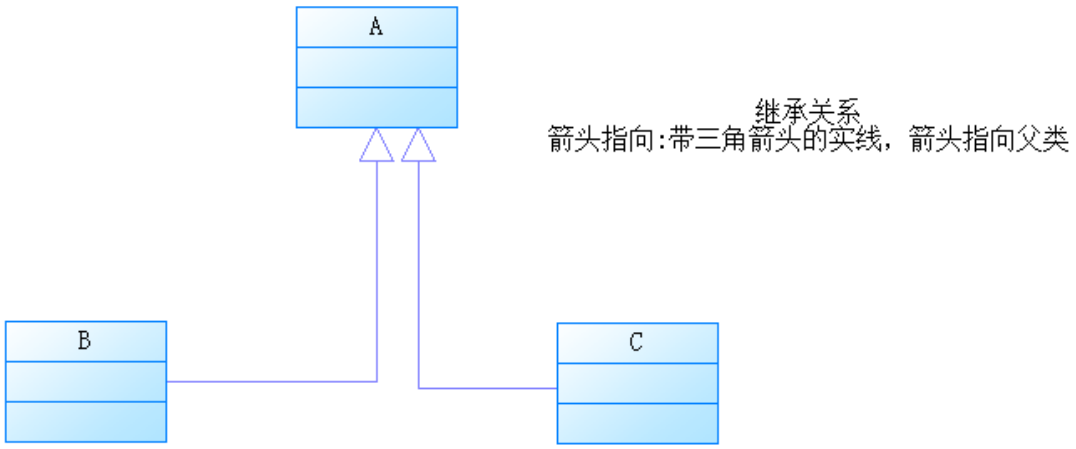
* 能够掌握类与类之间的的关系

  * 类与类之间的关系（泛化 >实现>组合> 聚合> 关联> 依赖）



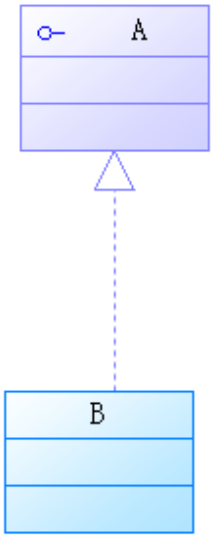
* 泛化关系: extends

    * 继承关系



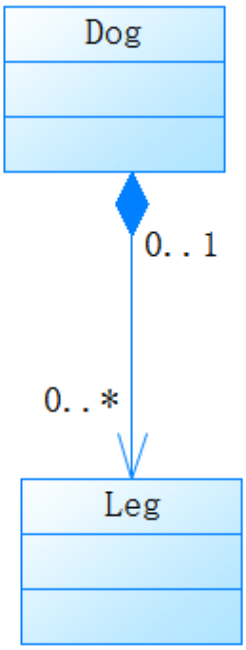
* 实现： implements

    * 实现接口



实现
箭头指向:带三角箭头的虚线，箭头指向接口

* 组合关系：关系对象出现在实例变量中
    * 整体与部分的关系, 但部分不能离开整体而单独存在
    * 组合是关联的一种
    * A类由若干B类组成,并且A类能决定B类的生命周期, 称A类为组合类
        * 狗（A类）拥有四条腿（B类）



组合关系
箭头指向:带实心菱形的实线,菱形指向整体

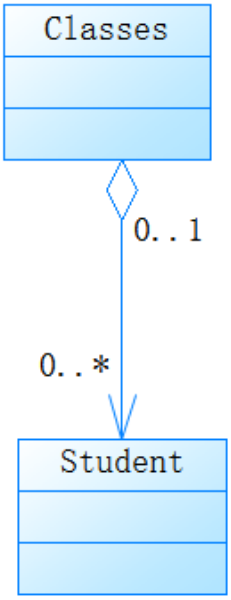
* 聚合关系:关系对象出现在实例变量中

    * 整体与部分的关系, 部分可以离开整体而单独存在

    * 聚合是关联的一种

    * A类由若干B类组成,但是A类不能决定B类的生命周期, 称A类为聚合类

        * 班级（A类）和班里学生（B类）



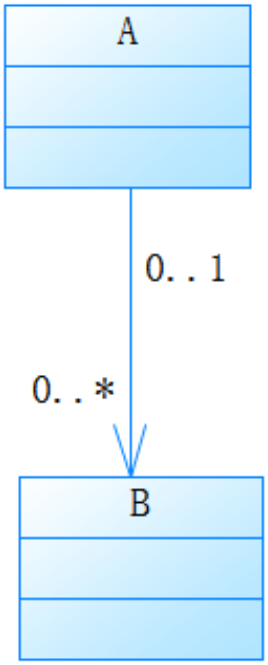
聚合关系
箭头指向:带空心菱形的实心线,
菱形指向整体

* 关联关系：关系对象出现在实例变量中

    * 一种拥有的关系

    * 一个对象A中持有对象B的引用

    * 代码中体现在一个类上声明一个成员变量

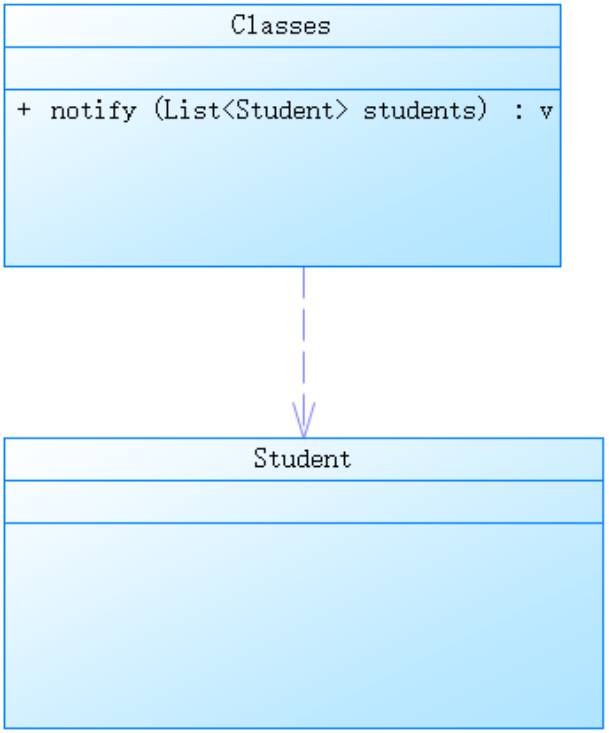


关联
箭头及指向:带普通箭头的
实心线，指向被拥有者

* 依赖关系：关系对象出现在局部变量或者方法的参数里，或者关系类的静态方法被调用

* 一种使用的关系, 即一个类的实现需要另一个类的协助

* 在A类的方法中, 使用B类作为方法的返回值类型,或者方法参数类型,或者局部变量类型, 称A类依赖B类



依赖

箭头指向：带箭头的虚线，指向被使用者