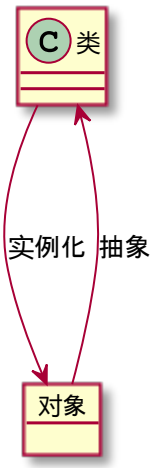
第1章 简介

1.1 面向对象基础

类与对象

类（Class）是一种面向对象计算机编程语言的构造，是创建对象的蓝图，描述了所创建的对象共同的属性和方法。而**对象**（Object）是类（Class）的一个具体实例（Instance）。例如小明养有一只名叫`小花`的狸花猫，这里`狸花猫`是类，小花是一只具体的猫，是对象。



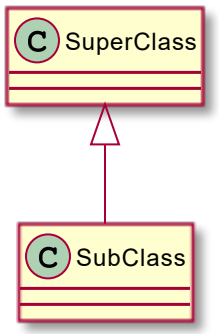
> Class: The definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures (known as class methods) themselves, i.e. classes contains the data members and member functions.
> Object: instances of classes

基本性质

封装（Encapsulation）包括以下内涵：

- 打包（绑定）属性和操作属性的方法；
- 对外隐藏内部数据，防止未授权的访问；
- 提供公有方法（getters、setters）实现属性访问。

继承（Inheritance）继承可以使得子类具有父类的属性和方法或者重新定义、追加属性和方法等。产生继承关系由泛化和特殊化两个过程产生。泛化（Generalization），又称广义化、一般化、通常化、普遍化、概念化，与之相对的是特殊化（Specialization）。通过概括将事物的定义进行修改或者补充以使其适用于更加大的范围，过程会伴随将该主体的定义或概念抽象化。一个将事物泛化的简单例子是归类。例如：将“大雁”归类为“鸟类”，将“鸟类”归类为“动物”。
继承关系的UML表达如下：



多态（Polymorphism）就是为不同类型实体提供统一的接口，或使用一个单一的符号来表示多个不同的类型。

> In programming languages and type theory, polymorphism is the provision of a single interface to entities of different types (Bjarne Stroustrup, 2007) or the use of a single symbol to represent multiple different types (Luca, 1985).

多态又分为**Ad-hoc多态**（Ad-hoc Polymorphism）、**参数化多态**（Parametric Polymorphism）和**子类型多态**（Subtyping Polymorphism）。Ad-hoc多态指一个多态函数可以应用于有不同类型的实际参数上，在Java等语言中体现为重载（Overloading）。例如下面代码中，add函数能适用于不同参数的多个实现：

```
public double add(double a, double b){
  return a + b;
}
public int add(int a, int b){
  return a + b;
}
```

参数化多态通过抽象符号对函数或数据类型进行泛化表达，使其支持统一操作并不依赖于具体的类型，在Java等语言中表现为泛型。例如JDK中的接口`java.util.List<E>`采用泛型定义，在使用过程中再给定类型：

```
List<String> strList1 = new ArrayList<String>();
List<Object> strList2 = new ArrayList<Object>();
```

子类型多态指相同的消息（这里消息传递或命令可以理解为函数调用）传递给不同的对象会引发不同的动作。在Java中，子类型多态需要继承、重写（Override）、类型转换（Casting）和动态绑定（Dynamic Binding）来实现。如下所示

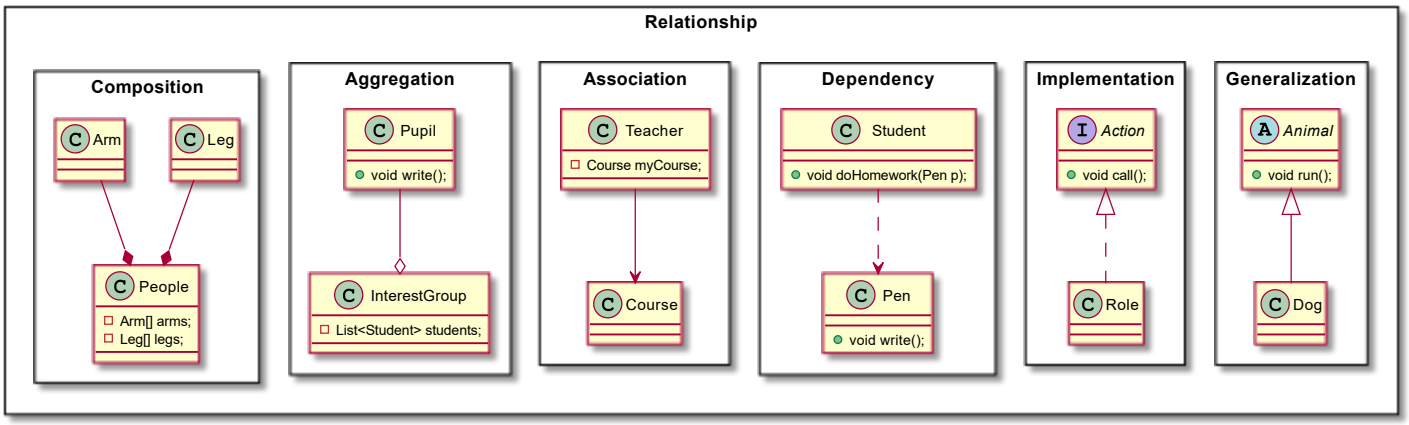
```
interface IAnimal{
  public void run();
}
class Dog implements IAnimal{
  public void run(){
    //Dog running
  }
}
class Cat implements IAnimal{
  public void run(){
    //Cat running
  }
}
class Test{
  public static void main(String[] args){
    IAnimal a = new Dog();
    a.run();
    a = new Cat();
    a.run();
  }
}
```

根据不同实现选择的时机，多态又分为静态多态性（又称编译时多态）和动态多态性（运行时多态）。Ad-hoc多态和参数化多态一般在编译时就确定具体的实现，而子类型多态一般在运行时确定具体实现。静态多态性程序执行效率更高，而动态多态性使得程序更加灵活。

1.2 类图中常见的关系

- **泛化关系**（Generalization）或称继承关系，表示一般与特殊的关系，指定了子类如何特化父类的所有特征和行为。
- **实现关系**（Implementation）是一种类与接口的关系, 表示类是接口所有特征和行为的实现。
- **依赖关系**（Dependency）表示一个类中会用到另一个类，类之间相对独立。

- **关联关系**（Association）是一种拥有的关系（has-a），它使一个类知道另一个类的属性和方法。
- **聚合关系**（Aggregation）是整体与部分的关系，且部分可以离开整体而单独存在。（whole-part, part can be shared）
- **组合关系**（Composition）是整体与部分的关系，但部分不能离开整体而单独存在。（whole-part, part can not be shared）



各类关系强度对比：泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖。

1.3 面向对象设计原则

《九阴真经》是金庸武侠世界中的顶级武学秘籍，其中上卷著有《总纲》一篇，根据道家思想总结出一些武学要旨，同样著有一些具体的武功招式。我们这里要学习的软件设计原则和设计模式就如同武学要旨与武功招式。面向对象设计原则可以看作为了达到某些软件设计目标（例如易读优雅、高复用、灵活可维护等）所需要遵循的一般规则，Martin首次在2000年介绍了五种原则：单一职责原则、开闭原则、里氏替换原则、接口隔离原则和依赖倒置原则，简称SOLID。

开闭原则

开闭原则指软件应该对扩展开放，对修改关闭。换句话说就是软件面对新需求能进行扩展，无需修改已有模块源代码。开闭原则是最核心的设计原则，其关键在于面向抽象编程思想。子类型多态和参数化多态都是遵循开闭原则的重要技术。

> The Open Closed Principle (OCP): A module should be open for extension but closed for modification. It means simply this: We should write our modules so that they can be extended, without requiring them to be modified.
>
> [`Principles and Patterns`]

为了理解开闭原则，下面设计了一个满足任意几何图形按面积排序的程序。

```
public class Test {
        public static void main(String[] args) {
                List<Geometry> list = new ArrayList<Geometry>();
                list.add(new Rectangle(2.0, 3.0));
                list.add(new Circle(1.0));
                Collections.sort(list);
                for(Geometry g:list) {
                        System.out.println(g.toString());
                }
        }
}
public abstract class Geometry implements Comparable<Geometry>{
        public abstract double getArea();
        @Override
        public int compareTo(Geometry o) {
                return (int) (this.getArea() - o.getArea());
        }
}
public class Circle extends Geometry{
        private double radius = 1.0;
        public Circle(double r) {
                radius = r;
        }
        @Override
        public double getArea() {
                return Math.PI * Math.pow(radius, 2);
        }
        public String toString() {
                return "Circle: " + getArea();
        }
}
public class Rectangle extends Geometry{
        private double width = 1.0;
        private double height = 1.0;
        public Rectangle(double w, double h) {
                width = w;
                height = h;
        }
        @Override
        public double getArea() {
                return width * height;
        }
        public String toString() {
                return "Rectangle: " + getArea();
        }
}
```

上述案例中，当需要增加三角形参与排序，只需要定义一个`Triangle`类，而不需要对其他类和排序函数`Collections.sort()`进行修改，完美地遵循了开闭原则。怎么做到的呢？这里同时用到了参数化多态和子类型多态，也充分体现了面向抽象编程。首先，追踪`Collections.sort()`函数的实现，会进入到`java.util.ComparableTimSort`类，在具体的排序算法中有两行关键代码：

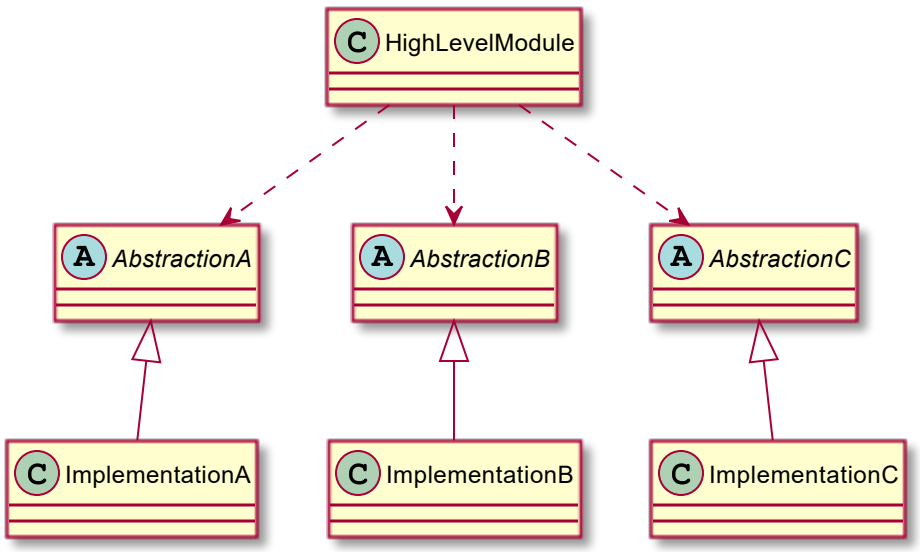
```
Comparable pivot = (Comparable) a[start];
pivot.compareTo(..)
```

这里看出排序的实现是依赖于`Comparable`这个接口的，或则说依赖于该接口的抽象方法`compareTo()`。无论对象排序需求如何改变，`Comparable`接口的定义是相对固定的，遵循了面向修改关闭，面对具体的需求，通过定义具体类实现该接口，又遵循了面向扩展开放。那么为什么这里的泛型主要用来保障类型安全和可扩展性。

依赖倒置原则

依赖倒置原则指高层模块不应该依赖底层实现模块，二者都应该依赖其抽象。在讨论开闭原则时已经接触了依赖倒置原则，如果说“开放扩展，关闭修改”是软件结构设计的目标，那么依赖倒置则是实现这一目标的主要机制。

> The Dependency Inversion Principle (DIP):Depend upon Abstractions. Do not depend upon concretions. If the OCP states the goal of OO architecture, the DIP states the primary mechanism. Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes. This principle is the enabling force behind component design, COM, CORBA, EJB, etc. [`Principles and Patterns`]



单一职责原则

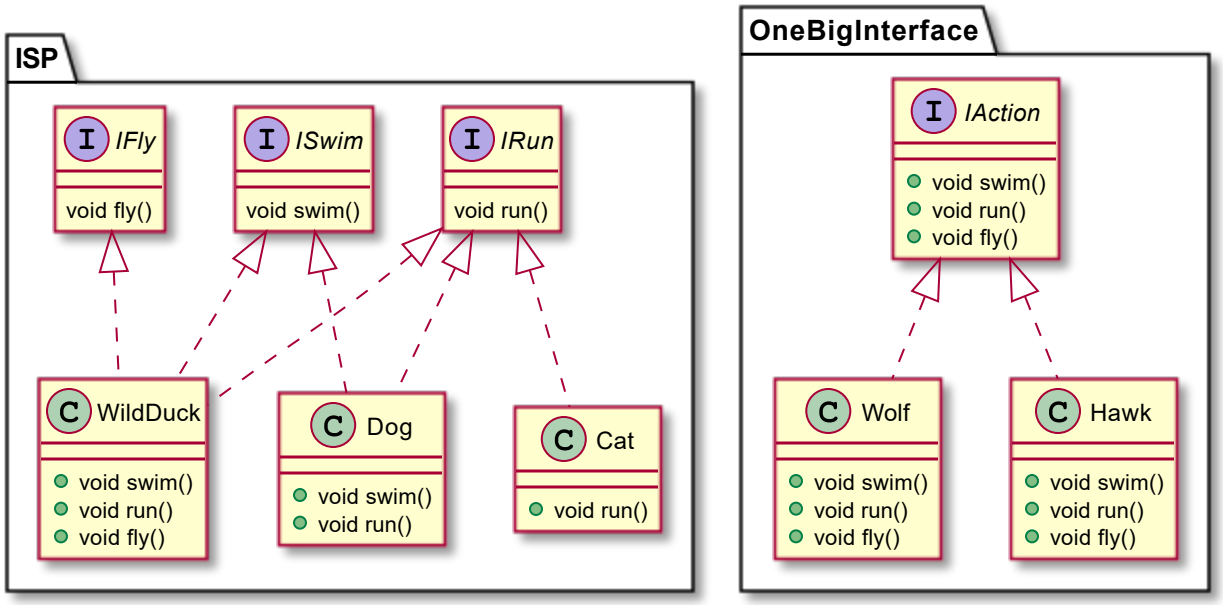
单一职责原则指一个类只存在唯一一个导致其变更的原因。

> The Single Responsibility Principle (SRP): A class should have one, and only one, reason to change.

接口隔离原则

接口隔离原则指一个类对另一个类的依赖应该建立在最小的接口上，客户端不应该依赖它不需要的接口。如下图所示，单一的接口 `IAction` 定义了动物的多种行为特征，当定义某种动物类时（例如狼和鹰），需要实现接口的所有函数，尽管一些行为是某类具体动物不具备的。

> The Interface Segregation Principle (ISP): Many client specific interfaces are better than one general purpose interface. [`Principles and Patterns`]



接口隔离原则和单一职责原则的区别在于：

- 单一职责原则原注重的是职责，而接口隔离原则注重对接口依赖的隔离；
- 单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口，主要针对抽象，针对程序整体框架的构建。

里氏替换原则

里氏替换原则指继承必须确保超类所拥有的性质在子类中仍然成立，即：所有引用基类的地方都能用子类对象替换，不影响业务逻辑。

> The Liskov Substitution Principle (LSP): Subclasses should be substitutable for their base classes. If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client and multiply inherit them into the class. [`Principles and Patterns`]

里氏替换原则可以解析为四个具体要求：

- 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。例如下面客户端中是否能将 `new SupClass();` 替换为 `new SubClass();` 。

```
class Test{
  public static void main(String[] args){
    SupClass c = new SupClass();  //Can be replaced with new SubClass()?
    c.add(1, 2);
  }
}
class SupClass{
  public double add(int a, int b){
    return a + b;
  }
}
class SubClass extends SupClass{
  public double add(int a, int b){
    return a - b;
  }
}
```

- 子类中可以增加自己特有的方法。扩展方法为新增方法，不涉及父类的应用场景。
- 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。重载不是重写，变量类型为父类，则调用父类的方法。这里主要考虑子类能顺利接替父类的业务。如下代码中，子类方法形参（List）比父类方法形参（ArrayList）更宽松，使其能顺利替换父类方法应用的代码：

```
class Test{
  public static void main(String[] args){
    ArrayList al = new ArrayList();
    A a = new A();  // 替换为: B b = new B();
    a.fun(al)  // 替换为: b.fun(al);
  }
}
class A{
  public void fun(ArrayList a){...}
}
class B extends A{
  public void fun(List a){...}
}
```

- 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。父类方法的返回值要参与系统其他计算，只有子类重写方法的返回类型保持不变或更严格才能实现类型兼容。如下代码中，由于B重写了A的`fun()`方法，`a.fun()`会调用A中的方法实现，返回类型为`HashMap`，若将B中方法的返回类型改成`Object`，则会产生类型匹配问题：

```
class Test{
  public static void main(String[] args){
    A a = new B();
    Map m = a.fun();
  }
}
class A{
  public abstract Map fun();
}
class B extends A{
  public HashMap fun(){...}  // 返回类型要比Map更严格
}
```

迪米特法则

迪米特法则（Law of Demeter, LoD）又称最少知道原则（Least Knowledge Principle, LKP），一个对象应该对其他对象保持最少的了解。类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

合成复用原则

合成复用原则（Composite Reuse Principle, CRP）指尽量使用对象组合或聚合方式实现代码复用，而不是继承关系。继承和组合是代码复用的两种主要途径，采用继承方式复用，父类的实现暴露给子类，称为白箱复用；而采用组合的方式复用避免了暴露具体实现，称为黑箱复用。（Design Patterns 1994）

> Composition over inheritance (or composite reuse principle) in object-oriented programming (OOP) is the principle that classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement the desired functionality) rather than inheritance from a base or parent class.

1.4 设计模式简介

根据克里斯多弗·亚历山大在《建筑模式语言》中的表述，模式表示一类问题以及该类问题的一般解决方案。同样，设计模式是开发人员在软件开发过程中面临的一般问题的解决方案。根据GoF在《Design Patterns》中的介绍，设计模式包含四个要素：模式名称（Pattern Name）、问题（Problem）、解决方案（Solution）和效果（Consequences）。模式名称应

当能够对问题、解决方案和效果的内涵进行概括。

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.(`Christopher Alexander`)