面向对象开发技术考前预习

面向对象程序设计

OOPL中的类和对象

类的定义

了解Java,C++等面向对象语言的对象定义语法

一些惯例

- 类名首字母大写
- 数据字段private
- 通过Accessor(Getter)/Setter/Is 访问数据字段

可见性修饰符

public,private,protected

Java和C++中，封装性由程序员确定

封装

一种数据隐藏技术，用户只能看到封装界面上的信息，对象内部对用户是不可见的。

如何实现？

1. 修改属性的可见性来限制对属性的访问

2. 为每个属性创建一对赋值(setter)和取值(getter)方法，用于对这些属性进行访问
3. 在setter和getter方法中，加入对属性的存取限制

熟悉Java的四种访问权限

- public 可以被任何包下的任何类访问
- protected 只能被同一包下的所有类或者子类访问
- private 只能由当前类的方法访问
- default 可以被同一包内的所有类访问(子类也必须在同一包下)

类的数据字段/类属性

Java同时包含无对象变量和无对象方法，称为类变量和类方法

类变量

- 类变量被一个类的所有实例共享
- 通过static修饰符创建共享数据字段
- 通过类名直接引用
- 在加载类时，执行静态块来完成初始化

类方法

- 不能访问非静态成员
- 不能使用this，super引用
- 析构和构造函数不能为静态成员

静态方法和实例方法

使用static修饰的方法称为静态方法，使用实例对象调用的方法叫做实例方法

静态方法	实例方法
使用类名调用	使用实例对象调用
可以访问静态成员	可以直接访问静态成员
不可以访问实例成员	可以直接访问实例成员
不能直接调用实例方法	可以直接访问实例方法、静态方法
调用前初始化	实例化对象时初始化

内部类(嵌套类)

将一个类的定义放在另一个类的定义内部

类的使用

类的使用的两种形式：

- 允引 client
- 继承

对象的创建

创建对象语法

```
C++
    PlayingCard * aCard = new PlayingCard(Diamond, 3);
Java, C#
    PlayingCard aCard = new PlayingCard(Diamond, 3);
```

对象数组的创建

在Java中，new仅创建数组，数组包含的对象必须独立创建

```
PlayingCard cardArray[ ] = new PlayingCard[13];
for (int i = 0; i < 13; i++)
    cardArray[i] = new  PlayingCard(Spade,i+1);
```

C++

```
PlayingCard *cardArray = new PlayingCard[13];
```

构造函数

初始化新创建的对象

```
类名 (参数列表)
{
具体代码
}
```

必须保证对象实例创建后，初始状态时明确的

- 构造函数是隐式调用的(也可以显式调用，在子类构造方法中调用父类的构造方法)
- 缺省构造方法(编译程序提供的缺省构造方法调用父类的却省构造方法)

C++创建对象相关语法

初始化器

用于对象成员初始化和派生类对基类的初始化

```
Class PlayingCard {
public:
PlayingCard (Suits is, int ir)
: suit(is), rank(ir), faceUp(true) { }
...
};
```

默认拷贝构造函数

因为当一个类没有自定义的拷贝构造函数的时候系统会自动提供一个默认的拷贝构造函数，来完成复制工作。

```
void main()
{
    Test a(99);
    Test b= a;
    //Test b ; b=a; 这样是错误的
}
```

初始化

内存划分:对象指针存放在栈内存中，实际的对象存储在堆内存中

内存操作：对类的属性进行操作的操作在对内存上进行

常量初始化

Java中常量用final关键字声明常量

常量可以在构造函数中初始化

C++中用const关键字声明常量

const和final的区别

- Final仅说明相关变量不会赋予新值，不能阻止对象内部对变量值进行改变
- const常量，不允许改变

析构函数

C++

在内存中开始释放对象时，自动调用析构函数

内存回收

C++ 在程序中显示的指定不再使用的对象，将其使用内存回收 delete关键字

Java C# Smalltalk 具有垃圾回收机制

时刻监控对象的操作，对象不再使用时，自动回收其所占内存。通常在内存将要耗尽时工作。

对象结构

对象同一：具有相同的标识

对象相等：两个对象的标识不同，但具有相同的值

对象引用和复制

```
b=a 引用

c=clone(a) 浅克隆
基本数据类型占不同空间，非基本数据类型指向同一个对象(影子clone)
d=deep_clone(a)深克隆

```

类对象和反射

元类

将类看作对象，该类必定是另一个特殊类的实例，这种特殊类叫做元类

优点：

- 概念上一致，只用一个概念就可以描述系统中的所有成分
- 使类称为运行时刻一部分有助于改善程序设计环境
- 继承的规范化，类和元类的继承采用双轨制

类对象

类本身也是一个对象。这个特殊的对象也有其属性和方法，我们称之为类属性和类方法

类对象是更一般的类（称为Class类）的实例。

类对象通常都包括类名称、类实例所占用内存的大小以及创建新实例的能力。

类对象操作

获取类对象

```
C++
typeinfo aClass=typeid(Avariable);
java
Class aClass=aVariable.getClass();
```

获取父类

```
Class parentClass = aClass.getSuperclass(); // Java
```

获取类名称

```
char * name = typeinfo(aVariable).name(); // C++
String internalName=aClass.getName();//Java
String descriptiveName=aClass.toString();
```

检测对象类

```
Child *c=dynamic_cast<Child *>(aParentPtr);
    if (c!=0){ ... }   //C++
dynamic_cast在C++中只适用于具有虚函数的类（多态类）

if (aVariable instanceof Child)  ...
if (aCalss.isInstance(aVariable)) ...  //Java
```

类加载器

ClassLoader主要用于加载类文件，利用反射（newInstance())生成类实例

```
ClassLoader c1=this.getClass.getClassLoader();

Class cls=c1.loadClass("com.rain.B")

B b=(B)cls.newInstance();
```

反射

程序可以访问、检测和修改它本身状态或行为的一种能力

Java反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法，对于生意一个对象，都能调用它的任一一个方法，这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制

Java反射API

java.lang包下

Class:标识一个正在运行的java应用程序中的类和结构

java.lang.relect包下

- Field类 代表类的成员变量
- Method类 代表类的方法
- Constructor类 代表类的构造方法
- Array类 提供了动态创建数组，以及访问数组的元素的静态方法

通过反射调用Field(变量)

- 获得当前类以及超类的public Field：
  - Field[] arrFields = classType. getFields();
- 获得当前类申明的所有Field：
  - Field[] arrFields = classType. getDeclaredFields();
- 获得当前类以及超类指定的public Field：
  - Field field = classType. getField(String name);
- 获得当前类申明的指定的Field：
  - Field field = classType. getDeclaredField(String name);
- 通过反射动态设定Field的值：
  - fieldType.set(Object obj, Object value);
- 通过反射动态获取Field的值：
  - Object obj = fieldType. get(Object obj) ;

得到某个对象的属性

```
1  public Object getProperty(Object owner, String fieldName) throws
   Exception {
2      Class ownerClass = owner.getClass();
3      Field field = ownerClass.getField(fieldName);
4      Object property = field.get(owner);
5      return property;
6  }
```

得到某个类的静态属性

```
1  public Object getStaticProperty(String className, String fieldName)
   throws Exception {
2      Class ownerClass = Class.forName(className);
3      Field field = ownerClass.getField(fieldName);
4      Object property = field.get(ownerClass);
5      return property;
6  }
```

执行某个对象的方法

```
public Object invokeMethod(Object owner, String methodName,
Object[] args) throws Exception {
    Class ownerClass = owner.getClass();
    Class[] argsClass = new Class[args.length];
    for (int i = 0, j = args.length; i < j; i++) {
        argsClass[i] = args[i].getClass();
    }
    Method method = ownerClass.getMethod(methodName,
argsClass);
    return method.invoke(owner, args);
}
```

执行某个类的静态方法

```
public Object invokeStaticMethod(String className, String
methodName,Object[] args) throws Exception {
    Class ownerClass = Class.forName(className);
    Class[] argsClass = new Class[args.length];
    for (int i = 0, j = args.length; i < j; i++) {
        argsClass[i] = args[i].getClass();
    }
    Method method = ownerClass.getMethod(methodName,
argsClass);
    return method.invoke(null, args);
}
```

新建实例

```
public Object newInstance(String className, Object[] args) throws
Exception {
    Class newoneClass = Class.forName(className);
    Class[] argsClass = new Class[args.length];
    for (int i = 0, j = args.length; i < j; i++) {
        argsClass[i] = args[i].getClass();
    }
    Constructor cons = newoneClass.getConstructor(argsClass);

    return cons.newInstance(args);
}
```

判断是否为某个类的实例

```
public boolean isInstance(Object obj, Class cls) {
  return cls.isInstance(obj);
}
```

得到数组中的某个元素

```
public Object getByArray(Object array, int index) {
    return Array.get(array,index);
}
```

消息传递

发送消息的流程

考虑对象A向对象B发送消息，也可以看成对象A向对象B请求服务

- 对象A要明确知道对象B提供什么样的服务
- 根据请求服务的不同，对象A可能需要给对象B一些额外的信息，以使对象B明确知道如何处理该服务
- 对象B也应该知道对象A是否希望它将最终的执行结果以报告形式反馈回去

消息传递语法

消息->接收器

```
aGame.displayCard(aCard,42,27)
```

接收器:aGame

选择器:displayCard

要求：aCard,42,27

伪变量:this

隐含指向调用成员函数的对象

类和继承

超类和子类

相关概念

超类：已存在的类

子类：新的类

子类不仅可以继承超类的方法，还可以继承超类的属性

子类可以重置超类中的方法，以便符合自己的需求

超类和子类的关系

子类 is a 超类

继承的传递性

超类的种类

- 直接超类
- 间接超类

单继承和多继承

单继承

一个类只有一个直接超类

单继承构成类之间的关系是一棵树

多继承

一个类有多于一个的直接超类

多继承构成类之间的关系是一个网格

C++中，多继承时，直接超类的构造函数调用顺序是

1. 若有多个抽象超类，按继承说明次序从左到右
2. 若有多个非抽象超类，按继承顺序从左到右

子类是否允许使用父类的属性和方案

子类可以调用父类的public,protected的属性和方案

泛化和特化

泛化

通过抽取及共享共同特征，将这些共性抽取出来作为超类放在继承层次的上端

抽取出来的超类叫做**抽象类**(abstract class)

- 抽象类一般没有实例
- 抽象类不能创建实例

特化

新类作为旧类的子类

接口与抽象类

接口可以继承于其他接口，甚至可以继承于多个父接口

抽象方法：介于类和接口之间的概念

- 定义方法但不实现
- 创建实例前，子类必须实现父类的抽象方法

替换原则

对于类A和类B，如果B是A的子类，那么在任何情况下都可以用类B来替换类A

可替换性是面向对象编程中一种强大的软件开发技术，它意味着：

变量声明时指定的类型不必与它所容纳的值类型相一致

分配方案

1. 最小静态空间分配：只分配基类所需的存储空间(C++)

- 对于指针(引用)变量，当消息调用可能被改写的成员函数时，选择哪个成员函数取决于接收器的动态数值
- 对于其他变量，关于调用虚拟成员函数的绑定方式取决于静态类,而不取决于动态类

2. 无论基类还是派生类，都分配可用于所有合法的数值的最大的存储空间

要求太严格，主要的OPPL中没有使用

3. 只分配用于保存一个指针所需的存储空间，在运行时通过对来分配数值所需的存储空间，同时将指针设为相应的合适值(Java)

- 堆栈中不保存对象值
- 堆栈中通过指针大小空间来保存标识变量，数据值保存在堆中
- 指针变量都具有恒定不变的大小，变量赋值时不会有任何问题

重置/改写

子类有时为了避免继承父类的行为，需要对其进行改写

语法上：子类定义一个与父类有着相同名称且类型签名相同的方法。

运行时：变量声明为一个类，它所包含的值来自于子类，与给定消息相对应的方法同时出现于父类和子类。

改写与替换结合时，想要执行的一般都是子类的方法。

重置overriding：可以重新修正从超类继承下来的属性及方法

重定义redefination：操作的表示和操作的实现都将改变

改写、重载、重定义

相同点：函数名相同，有不同的实现

区别：

- 发生在同一个类中，类型签名不同：重载
- 发生在父类和子类中，类型签名相同：改写
- 发生在父类和子类中，类型签名不同：重定义

改写的两种解释方式：

- 代替：操作子类实例时，父类的代码完全不会执行
- 改进：实现改进的方法将继承自父类的方法的执行作为其行为的一部分，这样父类的行为得以保留且扩充

延迟方法

父类中定义一个方法，但没有对其进行实现，我们称这个方法为一个延迟方法(抽象方法、纯虚方法),在静态类型面向对象语言中，对于给定对象，只有当编译器可以确认与给定消息选择器相匹配的响应方法时，才允许发送消息给这个对象

改写与遮蔽

改写区别于遮蔽的最重要的特征就是：

遮蔽是在编译时基于静态类型解析的，并且不需要运行时机制。

```
子类中关于 x 的声明将遮蔽父类中对同名变量的声明。
Class Parent {
Public int x = 12;
}
Class Child extend Parent {
Public int x = 42;   //sbadows variable from parent class
}

```

改写、遮蔽与重定义

1. 改写 父类与子类的类型签名相同，并且在父类中将方法声明为虚拟的。
2. 遮蔽 父类与子类的类型签名相同，但是在父类中并不将方法声明为虚拟的。
3. 重定义 父类与子类的类型签名不同。

C++隐藏机制

- 如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆，重载是在一个类之中的）
- 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。此时，基类的函数被隐藏（注意别与改写混淆）

父类可以通过final(java)，sealed(c++)关键字来防止子类对父类方法的改写

继承和构造函数

java中

- 父类构造函数不需要参数，父类构造函数和子类的构造函数都会自动地执行
- 父类构造函数需要参数时，子类必须显示地提供参数，用super关键字来实现

静态类和动态类

变量的静态类：指用于声明变量的类。静态类在编译时就确定下来，并且再也不会改变

变量的动态类：指与变量所表示的当前数值相关的类。动态类在程序的执行过程中，当对变量赋新值时可以改变

方法绑定

静态方法绑定/动态方法

相应消息时对哪个方法进行绑定是由接收器当前所包含的动态数值来决定的

Java中的实现继承

详情请看PPT上的例子,从中我们可以得出以下要点：

- 如果某个类具有一些只对类的部分对象适用的行为，那么不妨考虑将该类分裂为两个互相关联的类，两个类之间要么直接通过继承关联，要么通过一个公共的抽象类或接口关联。
- 可以使用继承、多态和动态方法调用来避免不优雅的条件判断。

软件复用机制

组合和继承

- 继承

通过继承，新的类可以声明为已存在类的子类

语义：A is-a B

- 组合

提供了一种利用已存在的软件组件来创建新的应用程序的方法。

语义：A has-a B

组合和继承的比较

继承

优点：

1. 子类可以重写父类的方法来实现对父类的扩展
2. 代码简洁性

缺点：

1. 父类的内部细节对子类是可见的
2. 子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为
3. 子类与父类是一种高耦合，如果对父类的方法做了修改的话（比如增加了一个参数），则子类的方法必须做出相应的修改

组合

优点：

1. 当前对象只能通过所包含的那个对象去调用其方法，所包含的对象的内部细节对当前对象不可见
2. 当前对象与包含的对象是低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码
3. 当前对象可以在运行时动态绑定所包含的对象

缺点：

1. 容易产生过多的对象
2. 为了能组合多个对象，必须仔细对接口进行定义

组合比继承更具灵活性和稳定性，所以在设计的时候优先使用组合

接口和抽象类

抽象类

定义方法时，可以只给出方法头，不必给出方法体，这样的方法叫做抽象方法

使用关键字abstract修饰(java),包含抽象方法的类必须声明为抽象类

抽象类不能实例化

子类必须实现其父类中所有的抽象方法，否则子类也必须声明为抽象类

抽象类中可以声明static属性和方法

作用：

- 代码重用
- 规划

应用

- 模板设计

接口

抽象类中只含有静态变量和抽象方法时，可以等同于一个接口。换句话说，接口就是特殊的抽象类

为什么使用接口?

- 多继承
- 设计和实现完全分离
- 更自然地使用多态

implements 关键字继承接口

接口特性

- 不可以被实例化
- 实体类必须实现接口的所有方法(抽象类除外)
- 可以实现多克接口
- 接口中的变量都是静态常量(java语法规定，接口中的变量自动隐含时public static final)
- 方法必须是public

面向接口编程

继承的形式

1. 特殊化继承

新类是基类的一种特定类型，它能满足基类的所有规范。

2. 规范化继承

保证派生类和基类具有某个共同的接口，即所有的派生类实现了具有相同方法界面的方法。

3. 构造继承

当继承的目的只是用于代码复用时，新创建的子类通常都不是子类型。称为构造子类化。

4. 泛化子类化

派生类扩展基类的行为，形成一种更泛化的抽象。

5. 扩展继承

如果派生类只是往基类中添加新行为，并不修改从基类继承来的任何属性，即是扩展继承。

6. 限制继承

如果派生类的行为比基类的少或是更严格时，就是限制继承

7. 变体子类化

两个或多个类需要实现类似的功能，但他们的抽象概念之间似乎并不存在层次关系。

8. 合并继承

可以通过合并两个或者更多的抽象特性来形成新的抽象。

一个类可以继承自多个基类的能力被称为多重继承 。

多态

多态的形式

1. 重载(专用多态)

类型签名

是关于函数参数类型、参数顺序和返回值类型的描述。

类型签名通常不包括接收器类型

2. 改写(包含多态)

层次关系中，相同类型签名

发生在有父类和子类关系的上下文中

3. 多态变量

指可以引用多种对象类型的变量

对于动态类型语言，所有的变量都可能是多态的

对于静态类型语言，则是替换原则的具体表现

多态设计思路

1. 编写父类
2. 编写子类，子类重写(覆盖)父类方法
3. 运行时，使用父类的类型，子类的对象

继承是子类使用父类的方法，而多态则是父类使用子类的方法。

多态两要素：

- 方法重写
- 使用父类类型

多态变量形式

- 简单变量

```
Animal pet;
pet = new Dog();
pet.speak();
```

- 接收器变量

多态变量最常用的场合是作为一个数值，用来表示正在执行的方法内部的接收器。

c++,java this

- 反多态(向下造型)

向下造型是处理多态变量的过程，并且在某种意义上这个过程的取消操作就是替换。

```
将不同的对象放入一个集合，取出时，如何知道对象的类型呢?
Child aChild
If (aVariable instanceof Child )
    aChild = ( Child ) aVariable
```

- 纯多态（ pure polymorphism ）

它支持代码只编写一次、高级别的抽象以及针对各种情况所需的代码裁剪。

通常，程序员都是通过给方法的接收器发送延迟消息来实现这种代码裁剪

```
关于纯多态的一个简单实例就是用JAVA语言编写的StringBuffer类中的append方法。这个方法的参数声明为Object类型，因此可以表示任何对象类型。
Class Stringbuffer{
    String append(Object value){
        return append(value.toString();}
    ...
}
```

- 方法toString在子类中得以重定义。
- toString方法的各种不同版本产生不同的结果。
- 所以append方法也类似产生了各种不同的结果。
- Append;一个定义，多种结果。

多态的运行机制

java多态机制时基于"方法绑定",就是建立method call（方法调用）和method body（方法本体）的关联。

先期绑定：绑定动作发生于程序执行前（由编译器和连接器完成） C编译器

后期绑定：执行期才根据对象型别而进行。后期绑定也被称为执行期绑定（run-time binding）或动态绑定（dynamic binding）。

Java中的所有方法，除了被声明为final的方法，都使用后期绑定

多态的一些细节

在运行时环境中，通过引用类型变量来访问所引用对象的方法和属性时，Java虚拟机采用以下绑定规则：

- 成员（实例）方法与引用变量实际引用的对象的方法绑定，这种绑定属于动态绑定，因为是在运行时由Java虚拟机动态决定的。
- 静态方法与引用变量所声明的类型的方法绑定，这种绑定属于静态绑定，因为实际上是在编译阶段就已经作了绑定。

- 成员变量（包括静态变量和实例变量）与引用变量所声明的类型的成员变量绑定，这种绑定属于静态绑定

java中，只有static和final方法是静态绑定的

动态绑定过程

1. 编译器检查对象的声明类型和方法名
2. 编译器检查方法调用中提供的参数类型
3. 当程序运行并且使用动态绑定调用方法时，**虚拟机必须调用同所指向的对象的实际类型相匹配的方法版本**。

4. 泛型

泛型是具有占位符（类型参数）的类、结构、接口和方法【T为委托类型】，这些占位符是类、结构、接口和方法所存储或使用的一个或多个类型的占位符。

泛型原理

Java泛型是在编译器的层面上实现的在编译后，通过擦除，将泛型的痕迹全部抹去。

泛型的好处

- 避免由于数据类型的不同导致方法或类的重载
- 类型安全
- 消除强制类型转换
- 多态的另一种表现形式。(参量多态)

重载和类型转换

重载

基于类型签名的重载

多个过程（或函数、方法）允许共享同一名称，且通过该过程所需的参数数目、顺序和类型来对它们进行区分。即使函数处于同一上下文，这也是合法的。

强制、转换(造型)

强制是一种隐式的类型转换，它发生在无需显式引用的程序中。

```
double x=2.8;
    int i=3;
x=i+x;//integer i will be  converted to real

```

转换表示程序员所进行的显式类型转换。在许多语言里这种转换操作称为“造型”。

```
x=((double)i)+x;
```

既可以实现基本含义的改变；也可以实现类型的转换，而保持含义不变

x是y的父类

```
x是y的父类
上溯造型
X a=new X();
Y b=new Y();
a=b; //将子类对象造型成父类对象，相当做了个隐式造型：a = (X)b;
下溯造型
X a=new X();
Y b=new Y();
X a1=b
Y b1=(Y)a1
```

作为转换的替换

强制：“类型的改变”，替换原则将引入一种传统语言所不存在的另外一种形式的强制。

发生在类型为子类的数值作为实参用于使用父类类型定义对应的形参的方法中时。

重载方法匹配算法

1. 找精确匹配（形参实参精确匹配的同一类型）找到，则执行，找不到转第二步。
2. 找可行匹配（符合替换原则的匹配，即实参所属类是形参所属类的子类），没找到可行匹配，报错；只找到一个可行匹配，执行可行匹配对应的方法；如果有多于一个的可行匹配，转第三步。
3. 多个可行匹配两两比较，如果一个方法的每个参数类型都与另一个方法的相应参数类型相同，或者是另一个方法相应参数类型的子类，那么这个方法就被认为是更具体的匹配。

UML类图

类图和对象图

在UML中类和对象模型分别由类图和对象图表示

类(Class)、对象(Object)和它们之间的关联是面向对象技术中最基本的元素

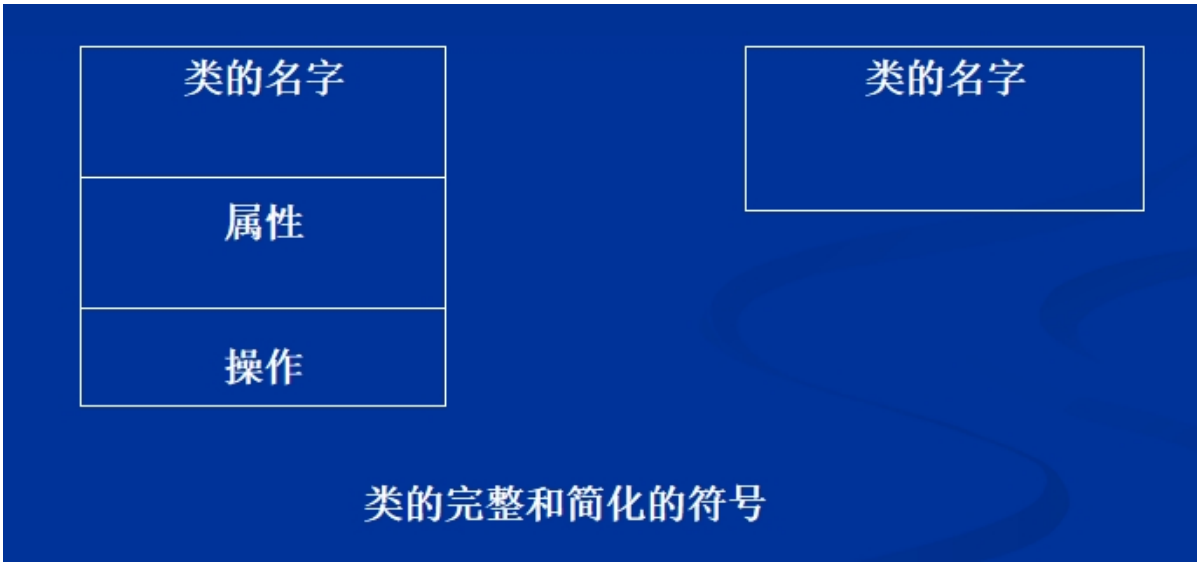
类图class diagram

- 描述类和类之间的静态关系
- 与数据模型不同
  - 显示了信息的结构
  - 还描述了系统的行为

- 类图是定义其它图的基础
- 在类图的基础上，状态图、交互图等进一步描述了系统其他方面的特性

类的表示

类描述一类对象的属性(Attribute)和行为(Behavior)



属性

- 描述该类对象的共同特点
- 属性表示关于对象的信息
- 只有系统感兴趣的特征才包含在类的属性中
- 系统建模的目的也会影响到属性的选取
- 通常对于外部对象来说，属性是可获取的，可设置的

属性的表示

[可见性]　属性名[多重性] [:类型][= 初始值]　[{特征串}]

可见性：＋ － #

多重性：表示属性的值可能有多个

特征串则是用户对该属性性质一个特征说明

类的操作

类操作的表示

[可见性] 操作名 [(参数表)] [: 返回类型][{特征串}]

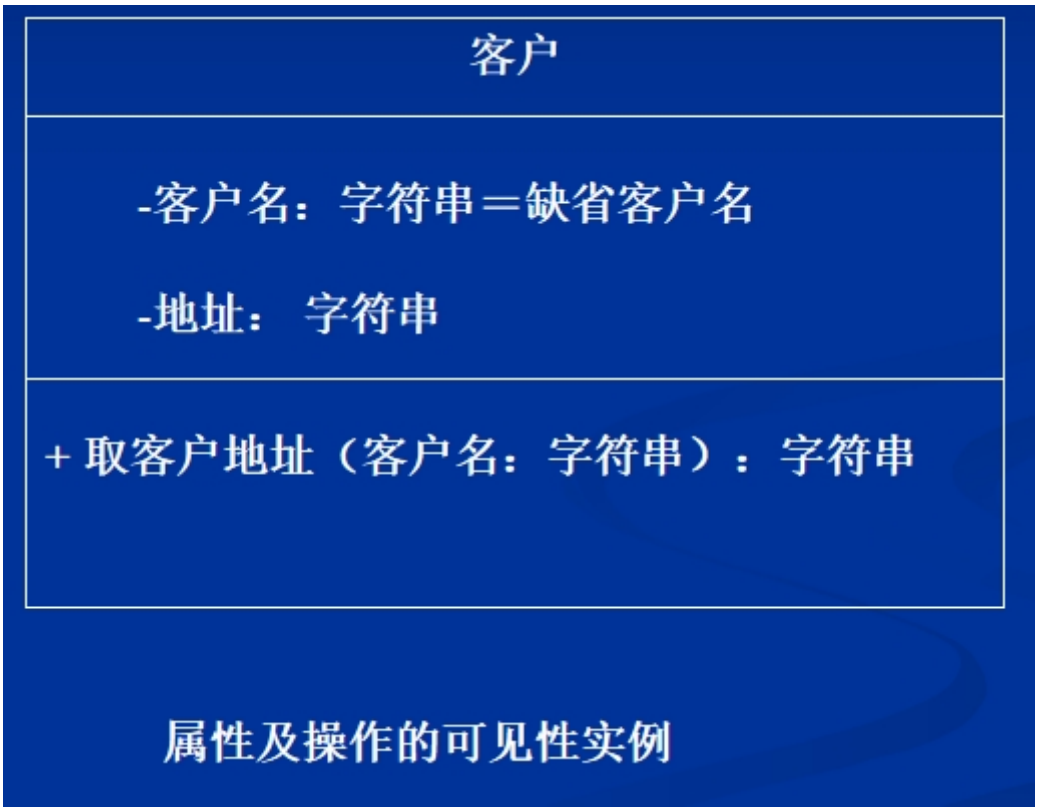
特征串主要有

查询(IsQuery)

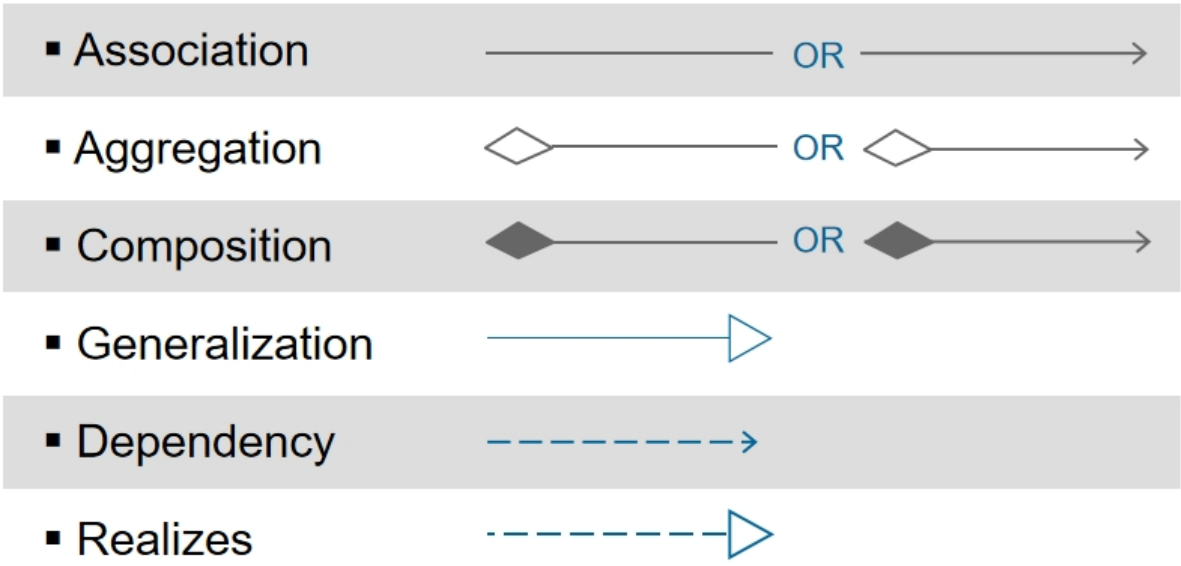
顺序(Sequence)

监护(guarded)和并发(concurrent)

其中后三个特征表达了操作的并发语义



类与类之间的关系



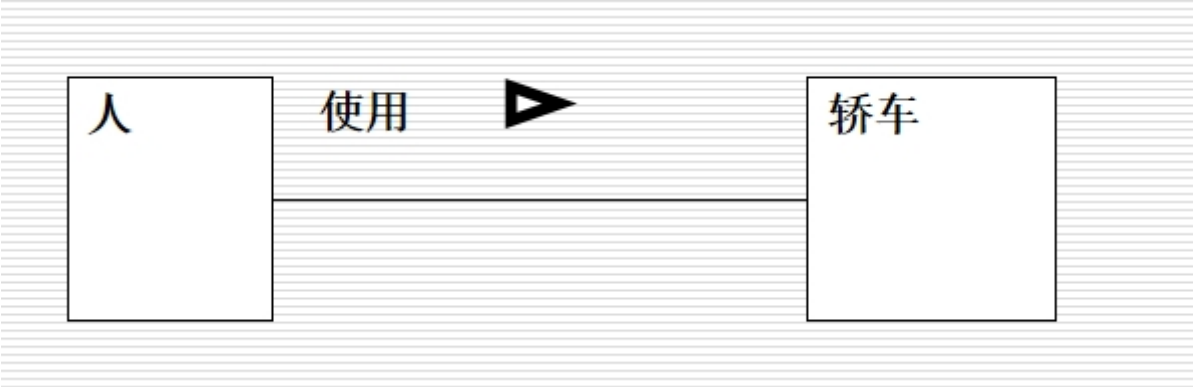
- 关联

表示两个或多个类或对象之间存在某种语义上的联系

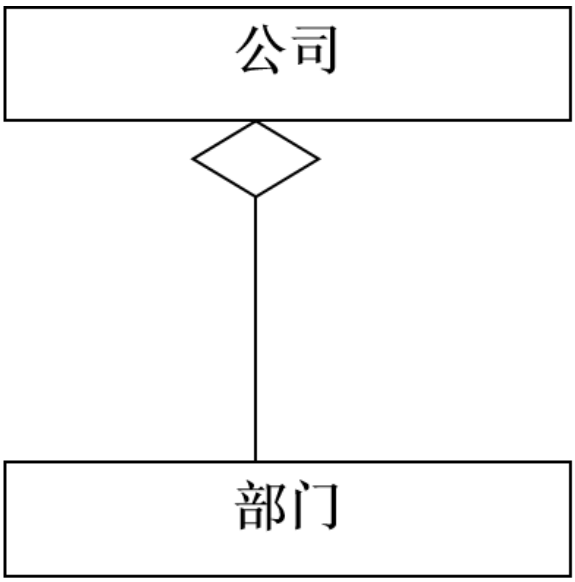
关联的方向：

- 关联上加上箭头表示方向，表示该关联单方向被使用
- 在UML中称为导航(Navigability)
- 将只在一个方向上存在导航表示的关联，称作单向关联（Uni-directional Association ）
- 在两个方向上都有导航表示的关联，称作双向关联 (Bi-directional Association)

不带箭头的关联:可以意味着未知、未确定或者该关联是双向关联三种选择



- 聚合关联



聚集对象和它的构成对象

构成对象不存在，聚集对象还可以存在

- 组成关联

表示整体拥有部分，整体不存在了，部分也会消失

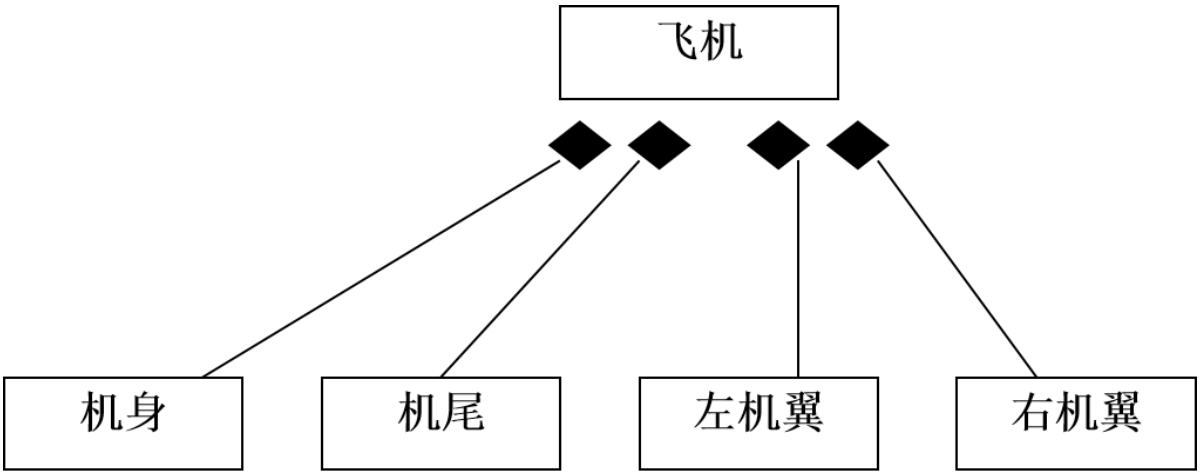


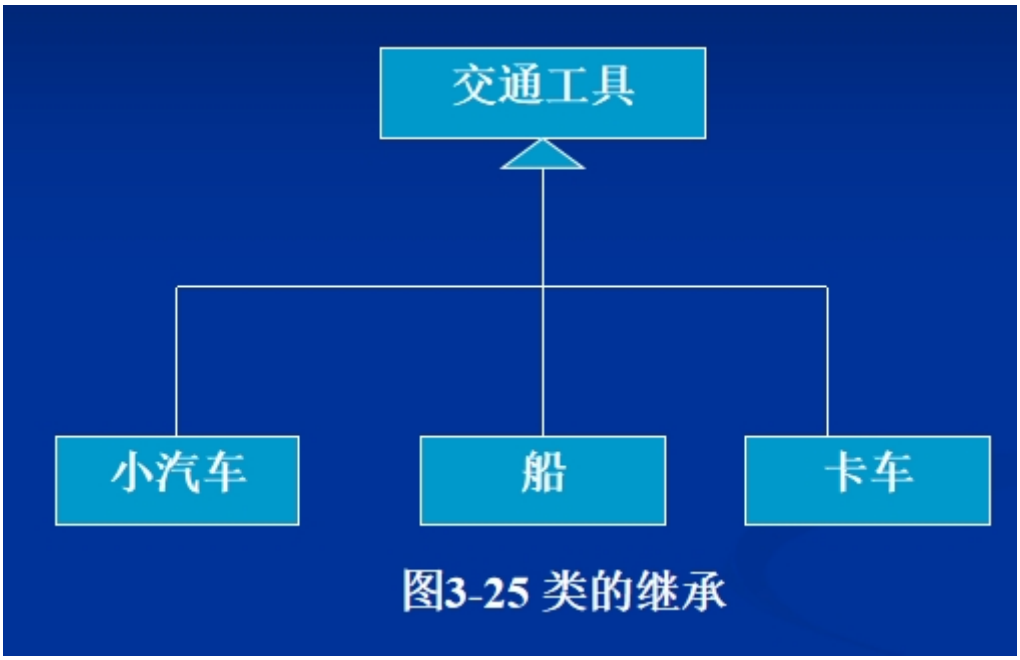
图3-23 一个组成对象和其成分对象

泛化及分类

一个类（一般元素）的所有特征（属性或或操作）能被另一个类（特殊元素）继承

泛化定义了一般元素和特殊元素之间的关系

继承的表示



抽象类

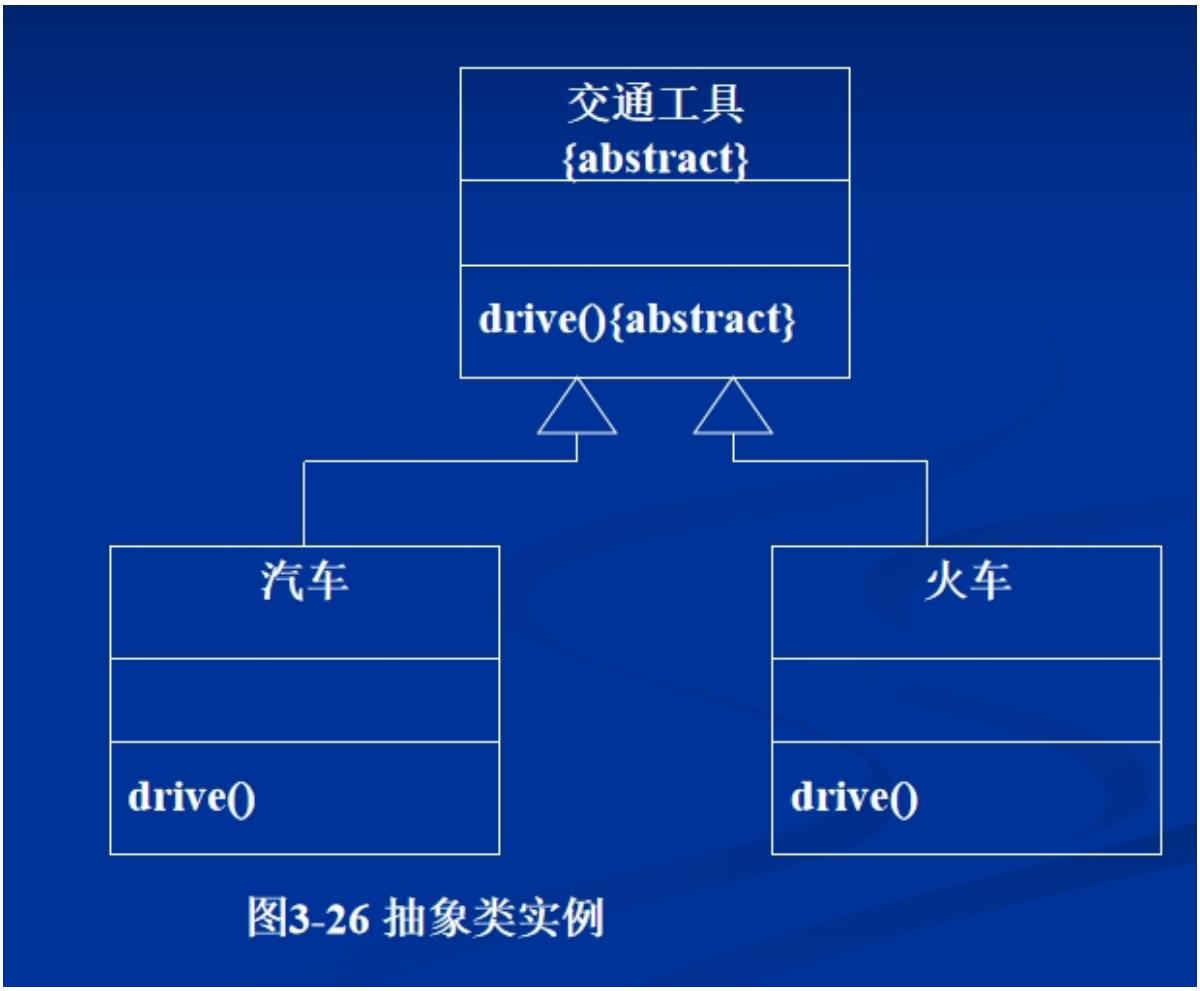
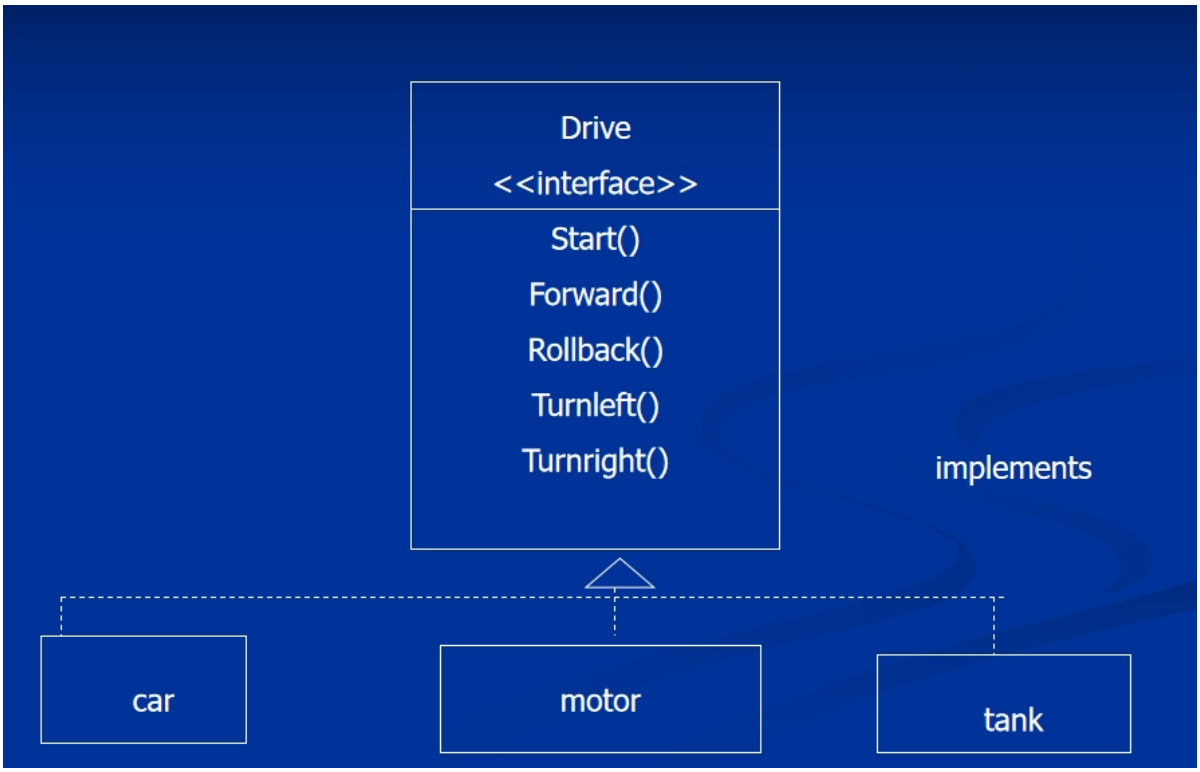


图3-26 抽象类实例

接口

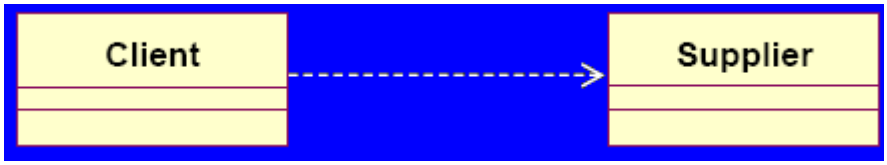


依赖

依赖是两个元素之间的关系

对一个元素（提供者）的改变可能影响或提供信息给其它元素（客户）

客户以某种方式依赖于提供者

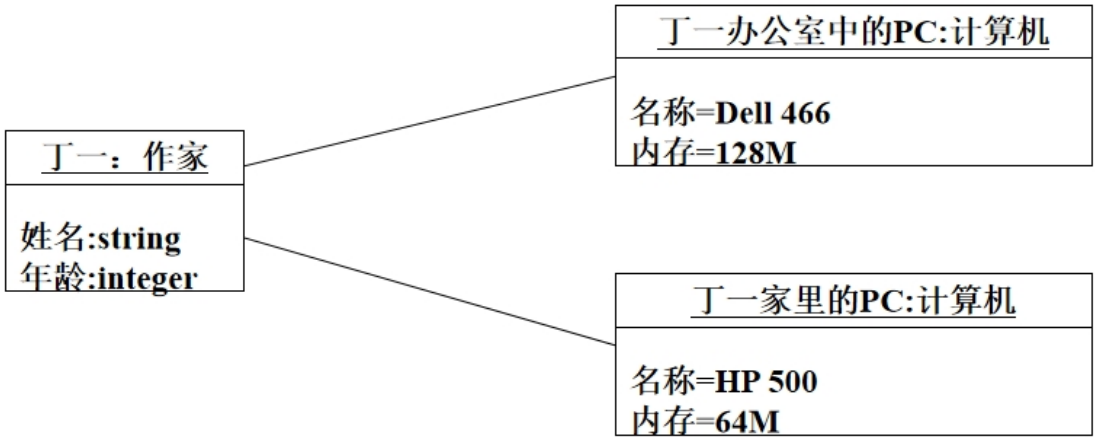


对象图

对象名：类名

：类名

对象名



(b) 对象图

面向对象设计模式