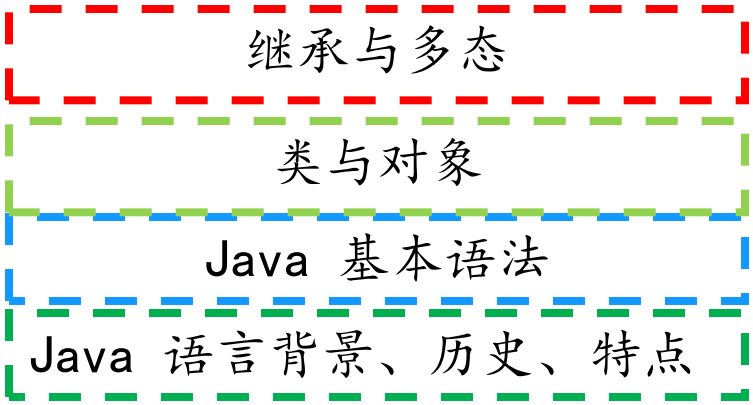
JAVA程序设计

内容提要



科技创新 引领未来
Scientific and technological innovation lead the future

第4章
继承与多态

内容提要

- 4.1 类的复用
- 4.2 多态
- 4.3 接口
- 4.4 内部类

面向对象的三个特性

◆基于面向对象的java语言具有更好的可扩展性和可维护性

这主要归功于java基于面向对象设计思想的三个主要特性，即:

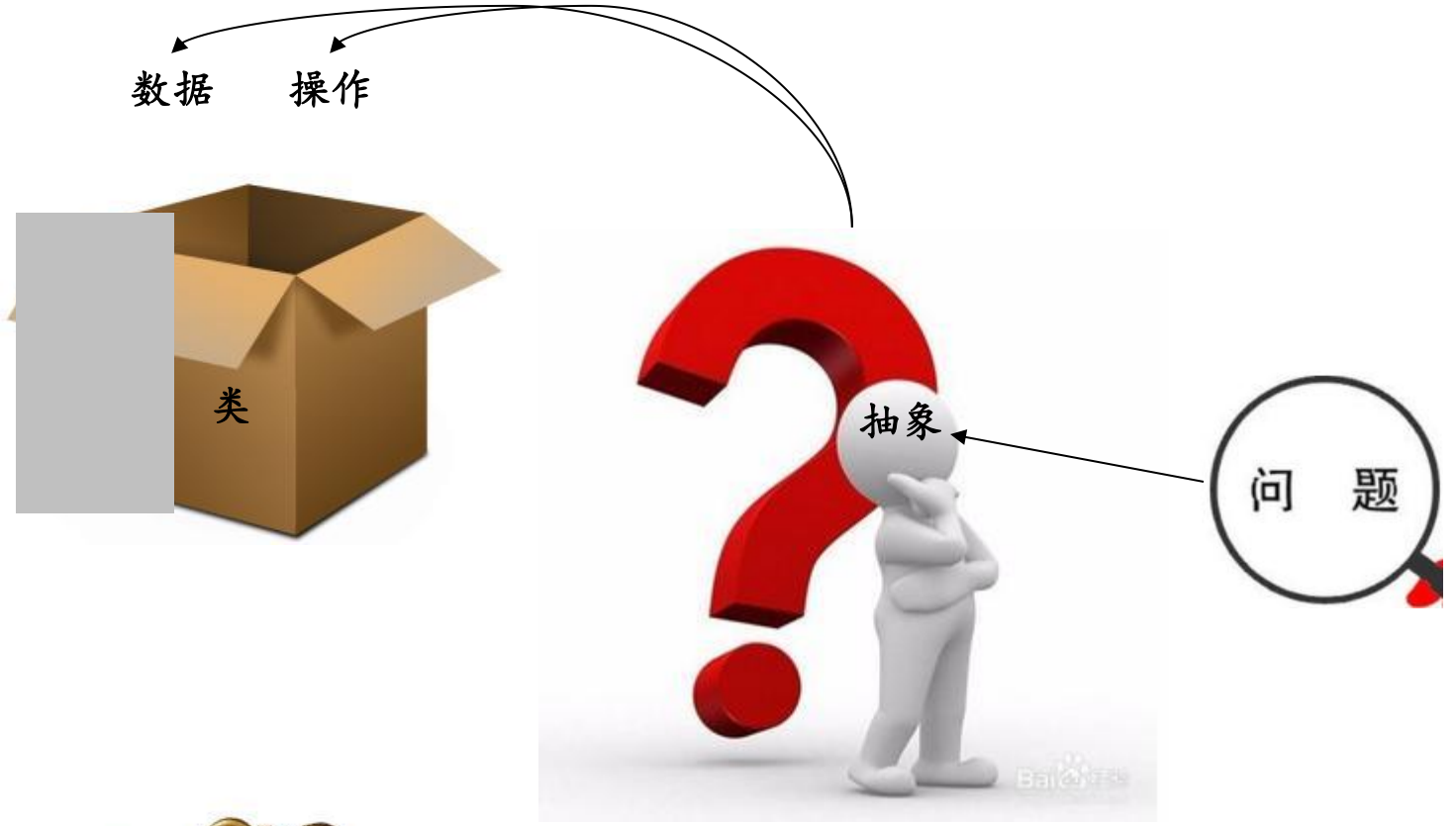
封装（encapsulation）
继承（Inheritance）
多态（Polymorphism）

●类（class）体现了面向对象设计思想中封装的特点。

●继承可以理解为人类社会中父与子的关系，父亲的一些特性可以遗传给儿子。

●多态是指若干个属于不同类的对象，对于同一个消息（如，方法调用）做出不一样的应答方式。

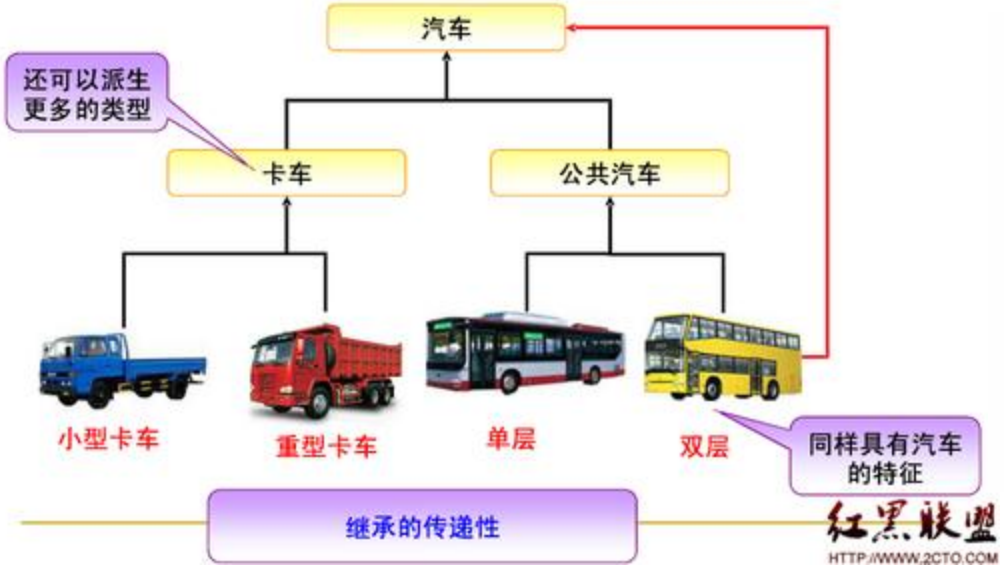
面向对象的三个特性



对象

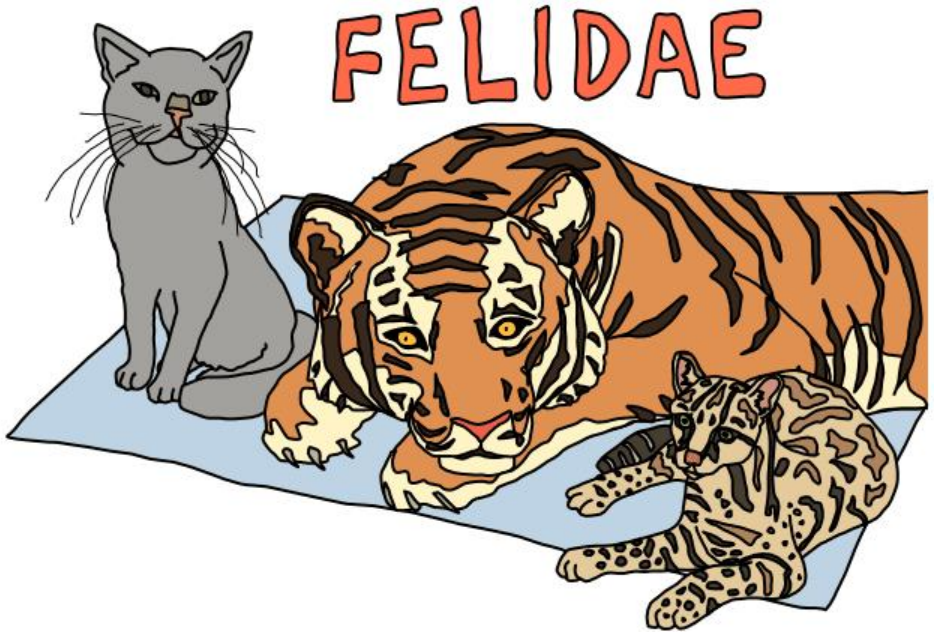
类（class）体现了面向对象设计思想中封装的特点。

面向对象的三个特性



●继承可以理解为人类社会中父与子的关系，父亲的一些特性可以遗传给儿子。

面向对象的三个特性



- 多态是指若干个属于不同类的对象，对于同一个消息（如，方法调用）做出不一样的应答方式。

面向对象的三个特性

◆基于面向对象的java语言具有更好的可扩展性和可维护性

这主要归功于java基于面向对象设计思想的三个主要特性，即:

封装（encapsulation）
继承（Inheritance）
多态（Polymorphism）

●类（class）体现了面向对象设计思想中封装的特点。

●继承可以理解为人类社会中父与子的关系，父亲的一些特性可以遗传给儿子。

●多态是指若干个属于不同类的对象，对于同一个消息（如，方法调用）做出不一样的应答方式。

4. 1类的复用Reusing Classes

➢Java作为面向对象语言的一个重要特点——可重用性

●可重用性可以从类定义的角度来体现。

- 类的概念封装了类,类的对象也就起到了代码复用的效果。

●还可以从类间关系的角度来考虑可重用性。

- 组合 (横向关系)
- 继承 (纵向关系)

4. 1类的复用

1.聚合/合成

2.继承

3.重写与重载

4. this和 final

5. abstract

4.1类的复用

1. 聚合/合成

- 在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分；
- 新的对象通过控制这些已有对象达到复用这些对象的目的，这即是类的聚合/合成。

4.1类的复用

```
class CentralProcessingUnit {
    private String type;

    CentralProcessingUnit() {
        System.out.println("Intel inside");
        type = "Core i3";
    }

    public String toString() {
        return type;
    }
}

public class ThinkPad {
    private String type, system, RAM, size;
    private CentralProcessingUnit CPU = new CentralProcessingUnit();
    private int i;
    private float f;

    public String toString() {
        return "type = " + type + " " + " system = " + system + " "
                + " RAM = " + RAM + " " + " size = " + size + "\n" + "i = " + i
                + " " + "f = " + f + " " + "CPU = " + CPU;
    }

    public static void main(String[] args) {
        ThinkPad laptop = new ThinkPad();
        System.out.println(laptop);
    }
}
```

```
Intel inside
type = null  system = null  RAM = null  size = null
i = 0 f = 0.0 CPU = Core i3
```

4.1类的复用

◆代码解释

上例中需要在ThinkPad类中使用CentralProcessingUnit类的功能，直接的方法就是在ThinkPad类中引用一个CentralProcessingUnit类的对象，这个ThinkPad类还组合了多个String类，因此ThinkPad类的功能是由多个类的功能组合在一起的。而基础数据类型（primitive）可以直接定义，它们的存在不属于组合。

◆toString()

每一个非基本数据类型的对象都拥有一个toString()方法。当编译器希望将一个对象以String类的方式表示时，该方法就会被激活。

在该例中，当使用“CPU = ” +CPU语句时，我们将得到一个String对象，而CPU是一个自己定义对象。 因此，编译器就会调用CPU所属CentralProcessingUnit类中的toString()方法。如果我们需要这种使用方法，就必须在定义类时重写toString()方法。

否则，在将对象输出为字符时，使用object类中的toString()方法，输出的是该对象的内存地址。

4.1类的复用

2. 继承

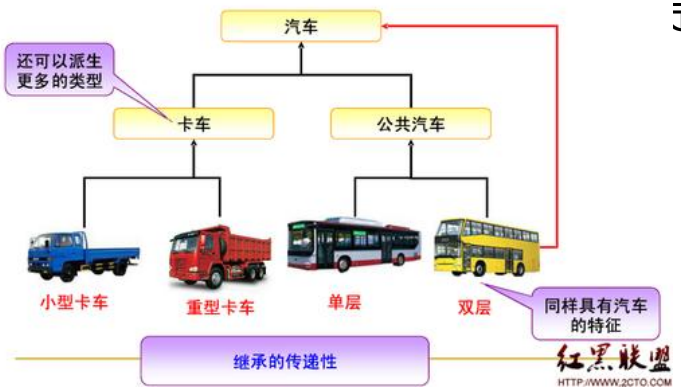
继承允许创建分层次的类，可以达到代码的复用，是面向对象编程技术的核心机制。

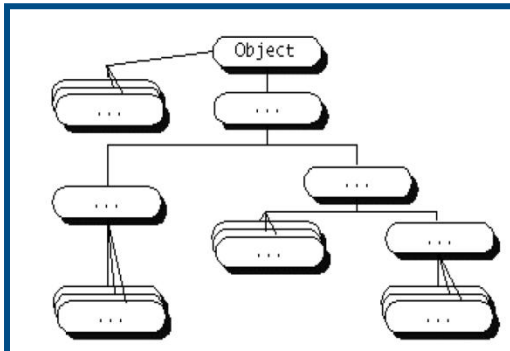
- 被继承的类叫超类（ superclass ）

  通过继承能够创建一个通用类，它可以被更具体的类继承。

- 继承超类的类叫子类（ subclass ）

  是超类的具体化，是超类的一个特定用途的类。





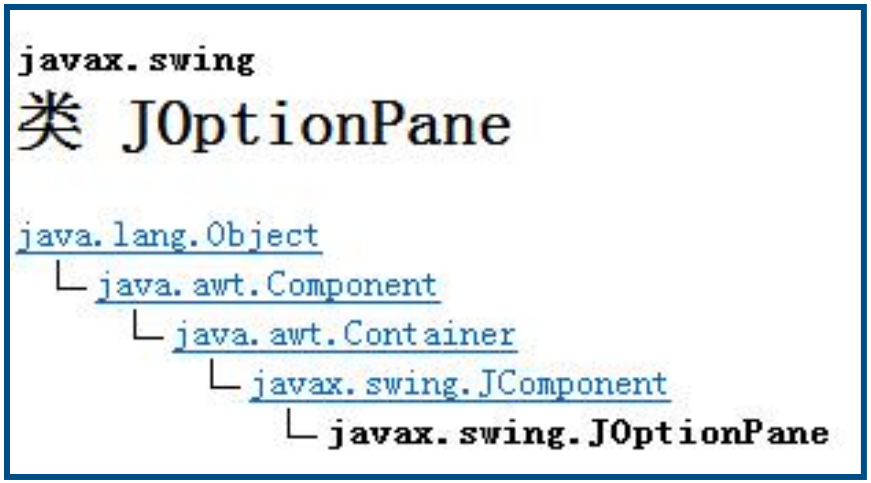
4.1类的复用

◆继承的语法

➢在新定义一个类继承已有类时，要使用extends关键字，其后跟上超类的名字，这样表示新定义的类是继承于超类。

声明一个继承超类的类的通常形式如下：

```
class subclass-name extends superclass-name {
// body of class
}
```



4.1类的复用

```
class SuperClass {
    int i, j;
    void printij() {
        System.out.println("i and j: " + i + " " + j);
    }
}
class SubClass extends SuperClass {
    int k;
    void printk() {
        System.out.println("k: " + k);
    }
    void printsum() {
        System.out.println("i+j+k: " + (i + j + k));
    }
}
public class SimpleInheritance {
    public static void main(String args[]) {
        SuperClass superOb = new SuperClass();
        SubClass subOb = new SubClass();
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.printij();
        System.out.println();
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.printij();
        subOb.printk();
        System.out.println();
        System.out.println("Sum of i, j and k in subOb:");
        subOb.printsum();
    }
}
```

```
Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

4.1类的复用

- 派生出来的子类进行继承时并不限于使用超类中原有的方法，可以在自己内部加入属于自己的新方法。
- 子类包括超类的所有成员，但它不能访问超类中被声明成private 的成员。
- 尽管superclass是subclass的超类，但它也是一个完全独立的类。

4.1类的复用

➢**Java继承的特征：**

●继承关系是传递的。

●继承简化了人们对事物的认识和描述，能清晰体现相关类间的层次结构关系。

●继承提供了软件复用功能。

●继承通过增强一致性来减少模块间的接口和界面，大大增加了程序的易维护性。

●Java出于安全性和可靠性的考虑，仅支持单重继承，而通过使用接口机制来实现多重继承。

4.1类的复用

◆如何初始化多层次的类？

- 子类并不仅仅是拥有了超类的属性和方法，在它的内部还包含了超类的一个“子对象”，这个“子对象”封装在了超类中。

- 需要在构造方法中完成这一过程，子类构造方法通过调用超类构造方法执行初始化。

- 在子类的构造方法中，Java会自动插入对超类构造方法的调用。

4.1类的复用

```
class Ball {
    Ball() {
        System.out.print("I hava a ball,");
    }
}

class Playing extends Ball {
    Playing() {
        System.out.print("I kick it");
    }

}

public class FootBall extends Playing {
    FootBall() {
        System.out.print("with my foot.");
    }

    public static void main(String[] args) {
        FootBall x = new FootBall();

    }
}
```

```
I hava a ball,I kick itwith my foot
```

```
class Ball {
    Ball(int i){

    }
}
```

```
class Ball {
    Ball(int i){

    }
    Ball(){

    }
}
```

构造方法的执行先于所有层次的类，所以超类会在子类访问它之前得到正确的初始化。

4.1类的复用

```
class Ball1 {
    Ball1(int x) {
        System.out.print("I have "+x+" Ball,");
    }
}

class Playing1 extends Ball1 {
    Playing1(String who) {
        super(2);
        System.out.print(who + "kick it");
    }
}

public class FootBall1 extends Playing1 {
    FootBall1() {
        super("John ");
        System.out.println(" with his foot.");
    }

    public static void main(String[] args) {
        FootBall1 x = new FootBall1();
    }
}
```

如果调用超类中含有参数的构造方法，必须明确地编写对超类的调用代码。这是用super关键字以及适当的一组自变量参数实现，注意super的位置。

```
I have 2 Ball,John kick it with his foot.
```

4.1类的复用

◆**super**是直接指向超类对象的变量，用来引用超类中的变量和方法。

```
class Country
   {
      String name;
      void value()
      {
         name="China";
      }
   }
```

```
class City extends Country
{
    String name;
    void value()
    {
        name="Beijing";
        super.value();
        System.out.println(name);
        System.out.println(super.name);
    }
public static void main(){
        City c = new City();
        c.value();
    }
}
```

结果：
Beijing
China

4.1类的复用

3. 重写与重载(Override&Overload)

➢重写（Override）

●有时子类并不想原封不动地继承父类的方法，而是需要作一定的修改，子类修改超类已有的方法叫重写。

- 超类方法的参数列表与被子类重写的方法的参数列表和返回类型必须相同。
- 被子类重写的方法不能拥有比超类方法更加严格的访问权限。访问权限大小关系为：

  private<默认<protected<public。
- 重写方法一定不会出现新的检查异常，或者比被重写方法声明更加宽泛的检查型异常。

4.1类的复用

```
class Person// 定义超类
{
    public void print() {// 超类中的方法
        System.out.println("超类Person的print方法!");
    }
}

class Student extends Person// 定义子类继承Person类
{
    public void print() {// 方法的重写
        System.out.println("子类Student的print方法!");
    }
}

public class OverrideExample01 {
    public static void main(String args[]) {
        Student s = new Student();
        s.print();
    }
}
```

```
子类Student的print方法!
```

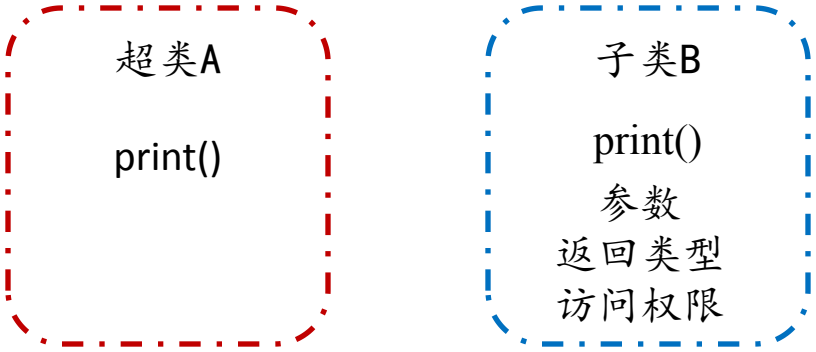
4.1类的复用

```
class Person
{   public void print() {    //public访问权限
    System.out.println( “超类Person的print方法! ”);
}
}
Class Stedent extends Person
{    private void print() {  //重写方法降低了访问权限，错误
     System.out.println( "子类Student的print方法!" );
}
}
```

超类的访问权限修饰符的限制一定不大于被子类重写方法的访问权限修饰符，private权限最小。如果某一个方法在超类中的访问权限是private，那么就不能在子类中对其进行重写，如果重新定义，也只定义了一个新的方法，不会达到重写的效果。

4.1类的复用

3. 重写与重载(Override&Overload)



4.1类的复用

➢重载(Overloading)

●所谓方法重载是指在一个类中，多个方法的方法名相同，但是参数列表不同。

- 在使用重载时只能通过不同的参数列表。
- 不能通过访问权限、返回类型、抛出的异常进行重载。
- 方法的异常类型和数目不会对重载造成影响。
- 可以有不同的返回类型。
- 可以有不同的访问修饰符。
- 可以显示出不同的异常。

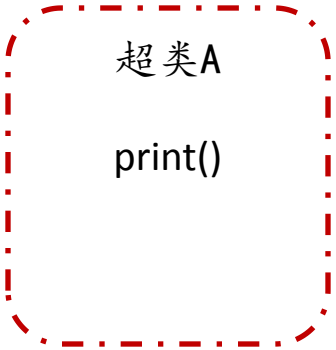
4.1类的复用

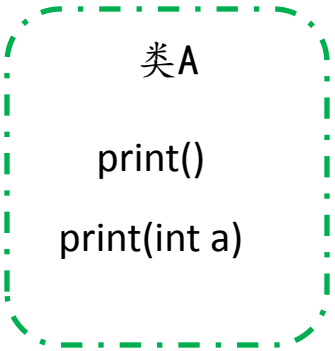
```
class Person1 {
    String name;
    int age;

    void print() {
        System.out.println("姓名: " + name + "年龄: " + age);
    }

    void print(String a, int b) {
        System.out.println("姓名: " + a + "年龄: " + b);
    }

    void print(String a, int b, int c) {
        System.out.println("姓名: " + a + "年龄: " + b + "ID号: " + c);
    }

    void print(String a, int b, double c) {
        System.out.println("姓名: " + a + "年龄: " + b + "ID号: " + c);
    }
}

public class OverLoadExample {
    public static void main(String args[]) {
        Person1 p1 = new Person1();
        p1.name = "李明";
        p1.age = 22;
        p1.print();
        p1.print("王小早", 19);
        p1.print("金波", 18, 100325);
        p1.print("婉宁", 25, 110903);
    }
}
```

```
姓名: 李明年龄: 22
姓名: 王小早年龄: 19
姓名: 金波年龄: 18ID号: 100325
姓名: 婉宁年龄: 25ID号: 110903
```

4.1类的复用

3. 重写与重载(Override&Overload)





4.1类的复用

➢方法重写与方法重载的区别

区别点	重载	重写（覆写）
英文	Overloading	Overriding
定义	方法名称相同，参数的类型或个数不同	方法名称、参数类型、返回值类型全部相同
	对权限没有要求	被重写的方法不能拥有更严格的权限
范围	发生在同一个类中	发生在继承类中

4.1类的复用

4. this和final

◆**this变量**使用在一个成员方法的内部，**指向当前对象**，即调用当前正在执行方法的那个对象。

```
class PersonInformation
      {
         String name,gender,nationality,address;
         int age;
         void PersonInformation(String p_name,
String p_nationality,String p_address,int p_age)
         {
            name=p_name;
            gender=p_gender;
            nationality=p_nationality;
            address=p_address;
            age=p_age;
         }
       }
```

```
class PersonInformation
         {
            String name,gender,nationality,address;
            int age;
            void PersonInformation(String name,String gender,
String nationality,String address,int age)
            {
            this.name=name;
            this.gender=gender;
            this.nationality=nationality;
            this.address=address;
            this.age=age;
            }
         }
```

4.1类的复用

◆this还有一个用法，就是**构造方法的第一个语句**，它的形式是**this(参数表）**，这个构造方法就会调用同一个类的另一个相对应的构造方法。

```
class UserInfo
{
    public UserInfo(String name)
    {
        this(name,aNewSerialNumber);
    }
    public Userinfo(String name,int number)
    {
        userName=name;
        userNumber=number;
    }
}
```

4.1类的复用

4. this和final

◆final修饰符

➢**final类**不能被继承，因此final类的成员方法没有机会被覆盖，默认都是final的。

- 英文翻译：最后的、最终的。
- 如果这个类不需要有子类，类的实现细节不允许改变，并且确信这个类不会再被扩展，那么就设计为final类。
- 如果一个类是完全实现的，并且不再需要子类继承，则它可以声明为final类。

注意事项

1.

```
final class FinalClass{};
class TestClass extends FinalClass{};
```

2.

```
final abstract class FinalClass2{};
```

4.1类的复用

➢final方法

如果一个类不允许其子类覆盖某个方法，则可以把这个方法声明为final方法。

- 是为方法“上锁”，防止任何继承类改变它的本来含义。

- 提高程序执行的效率。将一个方法设成final后，编译器就可以把对那个方法的所有调用都置入“嵌入”调用里。（内嵌机制）

4.1类的复用

➢final变量（常量）

●如果你希望你的变量不再被子类覆盖，可以把它们声明为final。

●用final修饰的成员变量表示常量，值一旦给定就无法改变。final修饰的变量有三种：静态变量、实例变量和局部变量，分别表示三种类型的常量。

●Blank final：final变量定义的时候，可以先声明，而不给初值。

- 无论什么情况，编译器都确保空白final在使用之前必须被初始化。

4.1类的复用

5. abstract

◆**abstract修饰符**表示所修饰的成分没有完全实现，还不能实例化。

●英文翻译：抽象、抽象的、摘要

●如果在类的方法声明中使用abstract修饰符，表明该方法是一个**抽象方法**，它需要在子类实现。

●如果一个类包含抽象方法，则这个类也是**抽象类**，必须使用abstract修饰符，并且不能实例化。

人

4.1类的复用

```
abstract class AbstractClass1
  {
     public int v1;
     abstract void test();
  }
  abstract class AbstractClass2 extends AbstractClass1
  {
     public int v2;
  }
  class NoAbstractClass extends AbstractClass2
  {
     void test() { }
  }
```

因为包含一个抽象方法test，类AbstractClass1必须被声明为抽象类。它的子类AbstractClass2继承了抽象方法test，但没有实现它，所以它也必须声明为抽象类。然而，AbstractClass2的子类NoAbstractClass因为实现了test，所以它不必声明为抽象的。

4.1类的复用

注意事项

1.

```
new AbstractClass1();
```

2.

```
class AbstractClass3
{
    abstract void test();
}
```

3.

构造方法

静态方法

final修饰的方法不能使用abstract修饰

4.

接口缺省为abstract

4.1类的复用

◆通过继承来进行复用

➢优点

●超类的大部分功能可以通过继承的关系自动进入子类，新类的实现、修改和扩展较为容易。

➢缺点

●继承将超类的实现细节暴露给子类，继承复用可能破坏类的封装性。

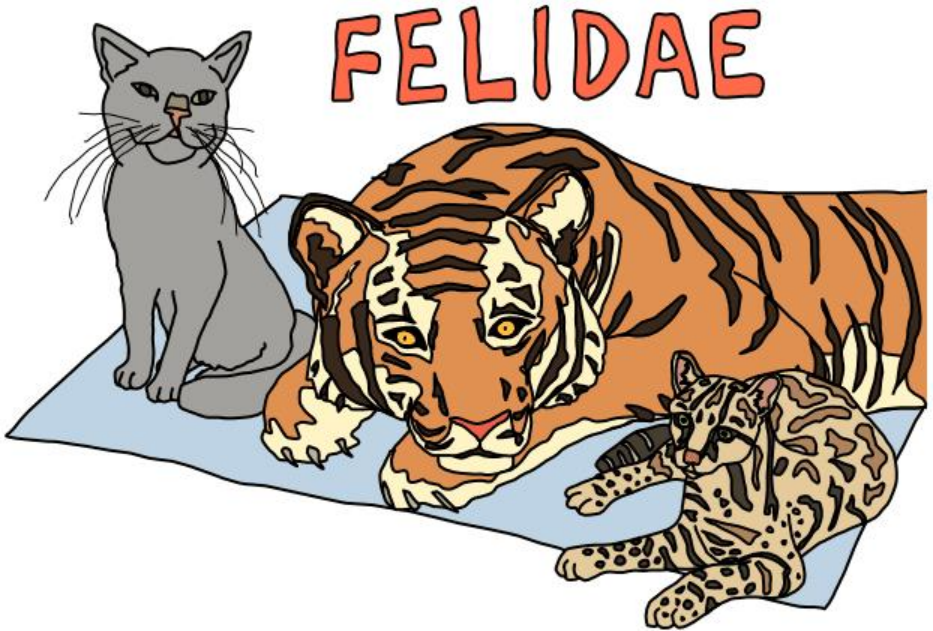
●从超类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性，只能在有限的环境中使用。

内容提要

- 4.1 类的复用
- 4.2 多态
- 4.3 接口
- 4.4 内部类

4.2多态

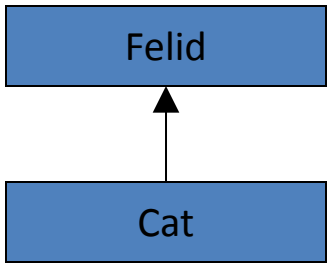
- polymorphism(多态)一词来自希腊语意为:“多种形式”。
- 做什么？怎么做？将接口与实现相分离
- 可读性、可扩展性
- 对象本身类型和超类类型



4.2多态

- Example1：Felid

```
class Felid {
    public void meow() {
        System.out.println("I am Felid!");
    }
    static void domeow(Felid f) {
        f.meow();
    }
}

public class Cat extends Felid{
    public static void main(String[] args) {
        Felid felid = new Cat();
        Felid.domeow(felid);
    }
}
```



4.2多态

- 向上转型

```
class Felid {
    public void meow() {System.out.println("I am Felid!");}
    static void domeow(Felid f) {f.meow();}
}
class Cat extends Felid{
    public void meow() {System.out.println("I am small cat, don't kick
my ass!");}
}
class Tiger extends Felid {
    public void meow() {System.out.println("I am Tiger, comes, enjoy
our dinner!");}
}
public class Test {
    public static void main(String[] args) {
        Cat felid1 = new Cat();
        Felid.domeow(felid1);
        Tiger felid2 = new Tiger();
        Felid.domeow(felid2);
    }
}
```

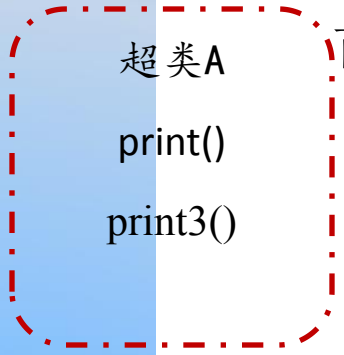
4.2多态

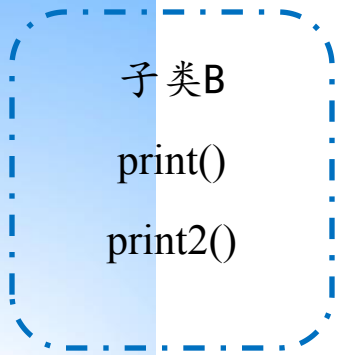
- 向上转型

```
class Felid {
   public void meow() {System.out.println("I am Felid!");}
   static void domeow(Cat c) {c.meow();}
   Statci void domeow(Tiger t) {t.meow();}

    Statci void domeow(Ocelot o) {o.meow();}
}
public class Test {
   public static void main(String[] args) {
      Cat felid1 = new Cat();
      Felid.domeow(felid1);
      Tiger felid2 = new Tiger();
      Felid.domeow(felid2);
      Ocelot felid3 = new Ocelot();
      Felid.domeow(felid3);
   }
}
class Ocelot extends Felid {
   public void meow() {System.out.println("I am Ocelot, I hat
Tiger!");}
}
```

4.2多态

- 向上转型
  - 所谓的向上转型，通俗地说就是子类转型成超类。通过定义一个超类类型的引用指向一个子类的对象，既可以使用子类的强大功能，又可以抽取超类的共性。





  - 超类类型的引用可以调用超类中定义的所有属性和方法，而对于子类中定义而超类中没有定义的方法，是无法使用的。
  - 超类中的一个方法只有在超类中定义而在子类中没有重写的情况下，才可以被超类类型的引用调用。
  - 对于超类中定义的方法，如果子类中重写了该方法，那么超类类型的引用将会调用子类中的这个方法，这就是动态绑定。

4.2多态

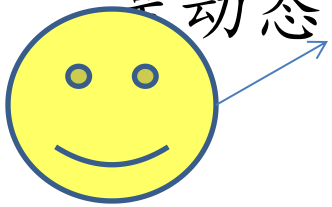
```
class Father {
    public void func1() {
        func2();
    }

    public void func2() {
        System.out.println("AAA");
    }
}

class Child extends Father {

    public void func1(int i) {
        System.out.println("BBB");
    }

    public void func2() {
        System.out.println("CCC");
    }
}

public class PolymorphismTest {
    public static void main(String[] args) {
        Father child = new Child();
        child.func1();
    }
}
```

这是个很典型的多态的例子。子类Child继承了超类Father，并重载了超类的func1()方法，重写了超类的func2()方法。重载后的func1(int i)和func1()不再是同一个方法，由于超类中没有func1(int i)，超类类型的引用child就不能调用func1(int i)方法；子类重写了func2()方法，那么超类类型的引用child在调用该方法时，将会调用子类中重写的func2()。

CCC

4.2多态

- 动态绑定（运行时绑定）
  - 绑定：将一个方法调用同一个方法主体关联起来被称为绑定。若绑定是在程序执行之前，被称为静态绑定。若绑定是在程序运行时决定的，那么称为动态绑定。
  - 发送消息给某个对象，让该对象去判定应该做什么事。
  - Java中的多态性是依靠动态绑定实现的。
  - Java中除了static方法和final方法，所有方法都是动态绑定的。



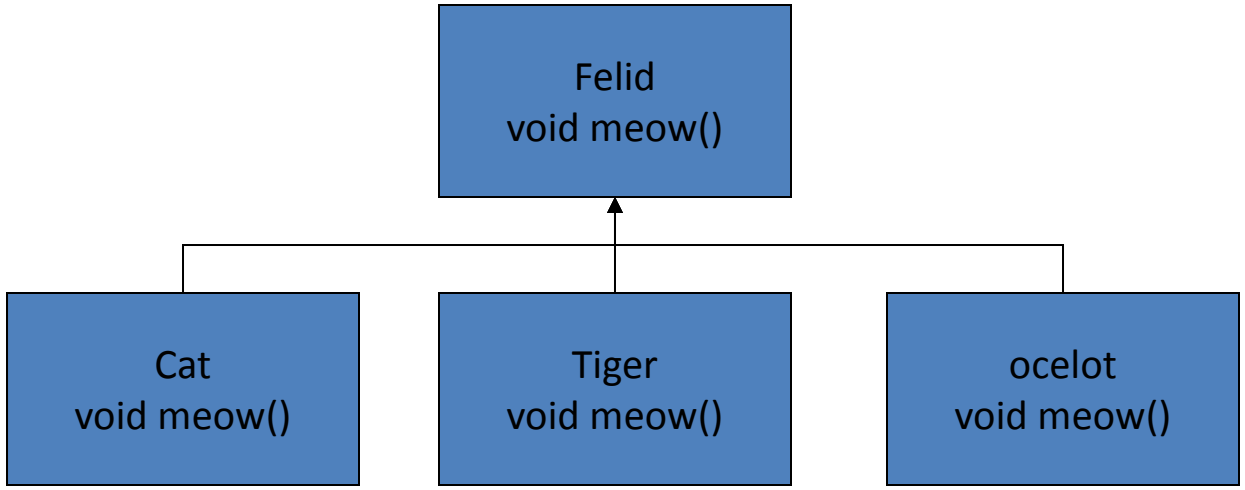
4.2多态

- 向上转型

```
class Felid {
    public void meow() {System.out.println("I am Felid!");}
    static void domeow(Felid f) {f.meow();}
}
class Cat extends Felid{
    public void meow() {System.out.println("I am small cat, don't kick
my ass!");}
}
class Tiger extends Felid {
    public void meow() {System.out.println("I am Tiger, comes, enjoy
our dinner!");}
}
public class Test {
    public static void main(String[] args) {
        Cat felid1 = new Cat();
        Felid.domeow(felid1);
        Tiger felid2 = new Tiger();
        Felid.domeow(felid2);
    }
}
```

4.2多态

- 可扩展性
  - 从前面的例子来看，由于多态机制，我们可以根据需要添加任意多个新类型，而不需要改变domeow方法。
  - 只与超类接口通信，这样的程序是可扩展的，因为可以从通用的超类继承出新的类型，从而新添一些功能。



4.2多态

- 可扩展性

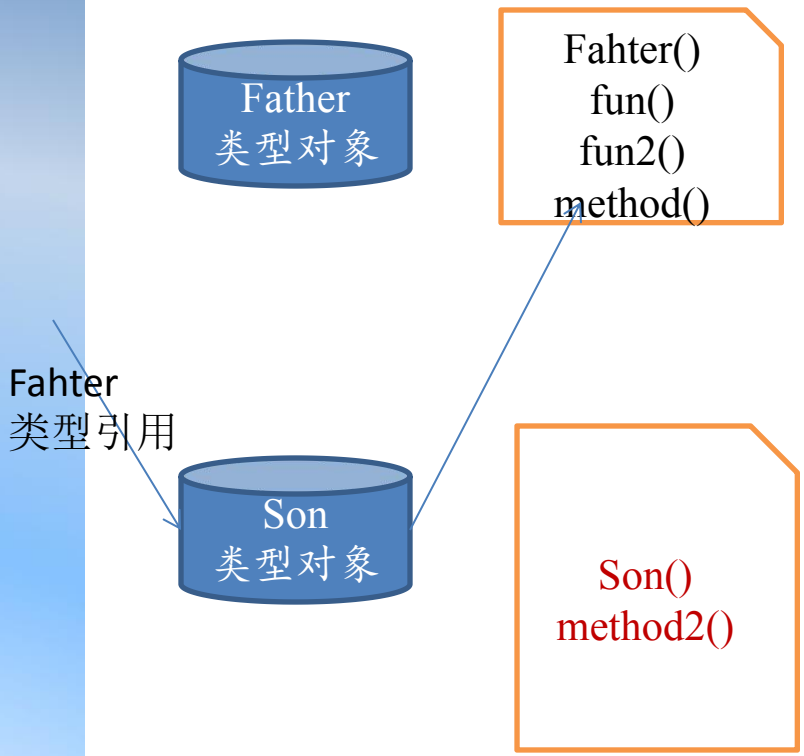
```
class Felid {
   public void meow() {System.out.println("I am Felid!");}
}
class Cat extends Felid{
   public void meow() {System.out.println("I am small cat, don't kick my ass!");}
}
class Tiger extends Felid {
   public void meow() {System.out.println("I am Tiger, comes, enjoy our dinner!");}
}
public class Test {
   public void static roar(Felid f) {
       f. meow();
   }
   public void static roarall(Felid [] e) {
       for(Felid i:e)
           roar(i);
   }
```

将改变的事物与未变的事物分离开来

```
   public static void main(String[] args) {
      Felid[] e = {
                    new cat();
                    new tiger();
      }
       roarall(e);
   }
}
```

4.2多态

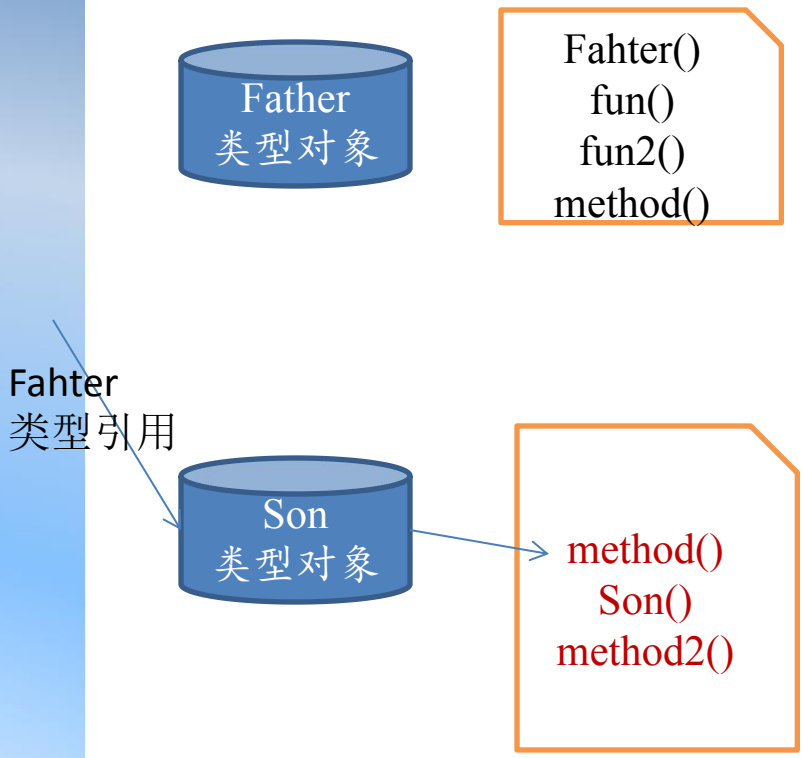
- 方法调用过程




```
class Father {
    public void method() {
        System.out.println("超类方法，对象类型："+this.getClass());
    }
    private void fun(){}
    public final void fun2(){}
    Fahter(){}
}
public class Son extends Father {
    public void static main(String[] args){
        Fahter sample = new Son();
        sample.method();
    }
    public void method2(){}
}
/*output:
超类方法，对象类型：class Son
*/
```

4.2多态

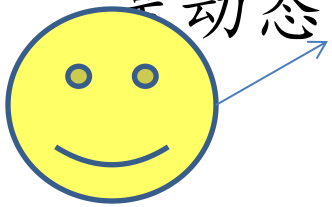
- 方法调用过程




```
class Father {
   public void method() {
      System.out.println("超类方法，对向类型："
"+this.getClass());
   }
   private void fun(){}
   public final void fun2(){}
   Fahter(){}
}
public class Son extends Father {
   public void method{
       System..out.println("子类方法，对向类型："
"+this.getClass());
   }
   public void static main(String[] args){
      Fahter sample = new Son();
      sample.method();
   }
   public void method2(){}
}
/*output:
子类方法，对向类型：class Son
*/
```

4.2多态

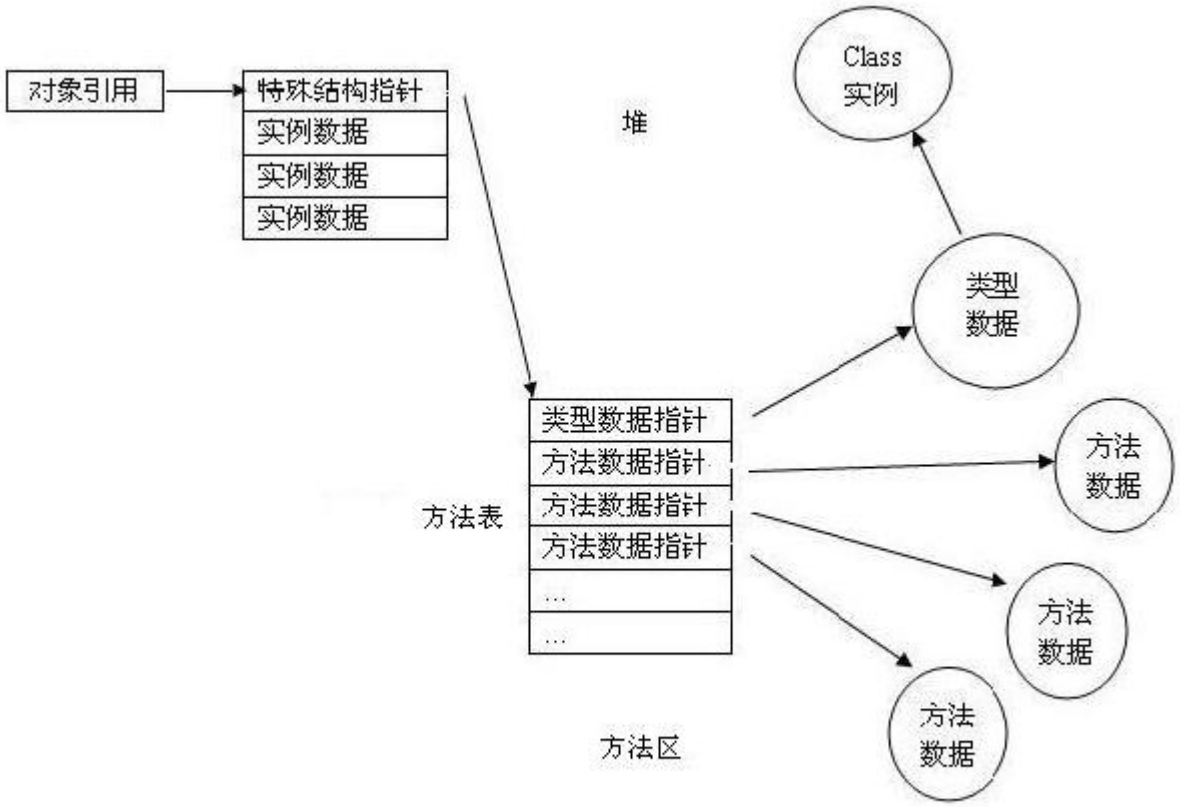
- 动态绑定（运行时绑定）
  - 绑定：将一个方法调用同一个方法主体关联起来被称为绑定。若绑定是在程序执行之前，被称为静态绑定。若绑定是在程序运行时决定的，那么称为动态绑定。
  - 发送消息给某个对象，让该对象去判定应该做什么事。
  - Java中的多态性是依靠动态绑定实现的。
  - Java中除了static方法和final方法，所有方法都是动态绑定的。



动态绑定为解决实际的业务问题提供了很大的灵活性，**是一种非常优美的机制。**

4.2多态

- 动态绑定（运行时绑定）

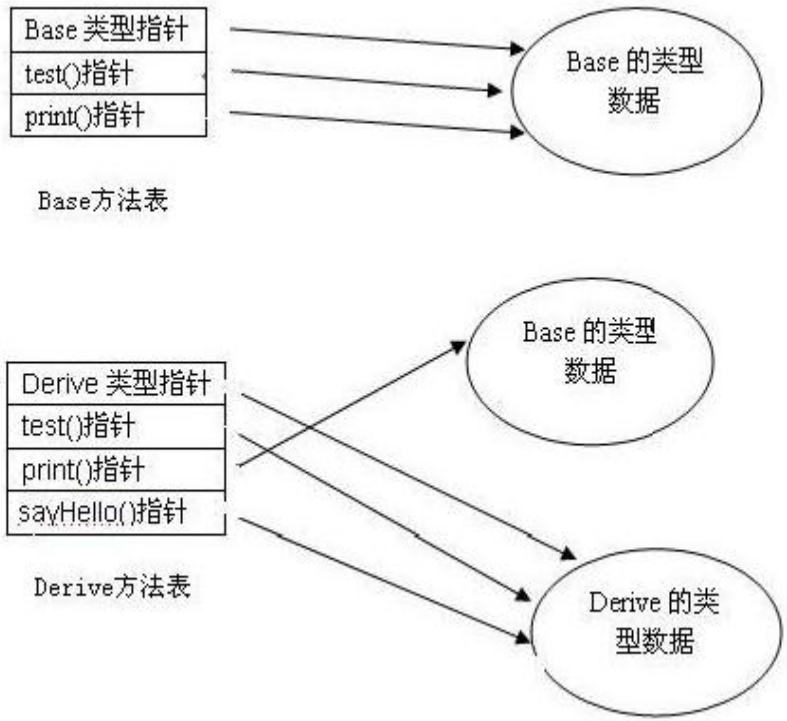


JAVA对象内存模型

4.2多态

- 动态绑定（运行时绑定）

```
public class Base
{
  public Base() { }
  public void test()  { System.out.println( "int Base" ); }
  public void print()  {  }
}

public class Derive extends Base
{
  public Derive()  {  }
  public void test() {System.out.println( "int Derive" ); }
  public void sayHello()  { }
  public static void main( String[] args )
  {
     Base base = new Derive();
     base.test();
  }
}
```



动态绑定为解决实际的业务问题提供了很大的灵活性，**是一种非常优美的机制。**

4.2多态

➢静态绑定变量成员

在处理 Java 类中的成员变量时，并不是采用运行时绑定，而是一般意义上的静态绑定。在向上转型的情况下，对象的方法可以“找到”子类，而对象的属性还是超类的属性。

```
class Father2 {
    protected String name="父亲属性";
    public void method() {
        System.out.println("超类方法，对象类型：" + this.getClass());
    }
}

public class Son2 extends Father2 {
    protected String name="儿子属性";

    public void method() {
        System.out.println("子类方法，对象类型：" + this.getClass());
    }
public static void main(String[] args) {
        Father2 sample = new Son2();  //向上转型
        System.out.println("调用的成员："+sample.name);
    }
}
```

```
调用的成员：父亲属性
```

这个结果表明，子类的对象(由超类的引用)调用到的是超类的成员变量。所以必须明确，**运行时（动态）绑定针对的范畴只是对象的方法。**

4.2多态

现在试图调用子类的成员变量name，该怎么做？最简单的办法是将该成员变量封装成方法getter形式，通过动态绑定方法来实现。

```
class Father3 {
    protected String name="父亲属性";
    public String getName() {return name; }
    public void method() {
        System.out.println("超类方法，对象类型：" + this.getClass());
    }
}

public class Son3 extends Father3 {
    protected String name="儿子属性";
    public String getName() {return name; }
    public void method() {
        System.out.println("子类方法，对象类型：" + this.getClass());
    }
public static void main(String[] args) {
        Father3 sample = new Son3();  //向上转型
        System.out.println("调用的成员："+sample.getName());
    }
}
```

```
调用的成员：儿子属性
```

如果此时把Father 中的getName方法去掉，编译时会出现"找不到符号getName“ 的错误。

4.2多态

- 私有方法

```
public class PrivateOverride {
   private void f() {
      System.out.println("private f()");
   }
   public void static main(String[] args){
      PrivateOverride po = new Derived();
      po.f();
   }
}
class Derived extends PrivateOverride {
   public void f() {
      System.out.println("public f()");
   }
}
/*output:
private f()
*/
```

4.2多态

- 向下转型
  - 向上转型是安全的。
  - 几何形状shape，圆？三角？矩形？
  - Java中所有转型都会得到检查。

```
class Useful{
   public void f() {}
   public void g() {}
}
class MoreUsefull extends Useful{
   public void f() {}
   public void g() {}
   public void u() {}
}
```

```
public class RTTI {
   public static void main(String[] args){
      Useful x = new Useful();
      Useful y = new MoreUseful();
       y.u();//Error,找不到方法
      ((MoreUseful)y).u();//Downcast
      ((MoreUseful)x).u();//Exception
   }
}
```

4.2多态

◆“polymorphism（多态）”一词来自希腊语，意为“多种形式”。

为了获得足够的功能，Java使用了多态性的概念，以达到面向对象语言所应有的功能。抽象类和接口是解决单继承规定限制和实现多态性的重要手段，多态是体现面向对象编程思想优势的重要特征。

4. 多态的实现 （3种形式）

➢继承实现的多态

●**方法的重写**:派生类（子类）将继承超类（父类）所有的方法、属性和事件，可根据需要来重新定义超类的方法、属性和事件，甚至增加或者修改部分内容，以提供不同形式的实现。

●**方法的重载:**是同一类中定义同名方法的情况。这些方法同名的原因，是它们的最终功能和目的都相同，但是由于在完成同一功能时，可能遇到不同的具体情况，所以需要定义含不同具体内容的方法，来代表多种具体实现形式。

4.2多态

- 可扩展性

```
class Felid {
   public void meow() {System.out.println("I am Felid!");}
}
class Cat extends Felid{
   public void meow() {System.out.println("I am small cat, don't kick my ass!");}
}
class Tiger extends Felid {
   public void meow() {System.out.println("I am Tiger, comes, enjoy our dinner!");}
}
public class Test {
   public void static roar(Felid f) {
       f. meow();
   }
   public void static roarall(Felid [] e) {
       for(Felid i:e)
           roar(i);
   }
```

将改变的事物与未变的事物分离开来

```
   public static void main(String[] args) {
      Felid[] e = {
                    new cat();
                    new tiger();
      }
       roarall(e);
   }
}
```

4.2多态

➢抽象类实现的多态

●使用abstract类型修饰符的抽象方法，属于一种不完整的方法。它们只含有一个声明，没有方法主体，超类不能实现它，必须由派生类重写实现，为其它子孙类用抽象机制实现多态性提供了统一的界面。

●创建抽象类和方法有时对我们非常有用，因为它们使一个类的抽象变成明显的事实，可明确告诉用户和编译器自己打算如何用它，每一个实现抽象类的子类都可以对抽象的方法进行不同的实现。

4.2多态

- 可扩展性

```
abstract  class Felid {

    public void abstract meow();
}
class Cat extends Felid{
   public void meow() {System.out.println("I am small cat, don't kick my ass!");}
}
class Tiger extends Felid {
   public void meow() {System.out.println("I am Tiger, comes, enjoy our dinner!");}
}
public class Test {
   public void static roar(Felid f) {
       f. meow();
   }
   public void static roarall(Felid [] e) {
       for(Felid i:e)
           roar(i);
   }
```

将改变的事物与未变的事物分离开来

```
public static void main(String[] args) {
   Felid[] e = {
                new cat();
                new tiger();
   }
    roarall(e);
 }
}
```

4.2多态

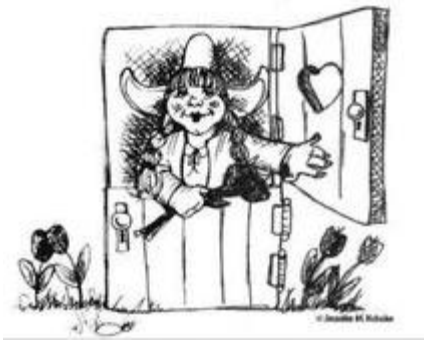
➢接口实现的多态

- “interface”（接口）关键字使接口的抽象概念更深入了一层，我们可将其想象为一个“纯”抽象类。
- 接口中只有方法声明，实现接口时无成员变量名字冲突问题，也没有对超类方法的重定义问题，也不存在重复继承问题，比一般类的多重继承简单。
- 弥补了Java中“一个子类，只能有一个超类”的不足，实现了多态性的“一个接口，多种方法”。

4.3 接口interface

➢接口的随意可插入性:

Java是一种单继承的语言，若要给已有超类的子类增加新功能，在OCP“开闭原则”原则下,解决是给它的超类加超类，或者给它超类的超类加超类，直到移动到类等级结构的最顶端。这样一来，对一个具体类的可插入性的设计，就变成了对整个等级结构中所有类的修改。而有了接口,以上例子中,就不需要维护整个等级结构中的所有类了。



“软件实体应当对扩展开放，对修改关闭”

4.3 接口

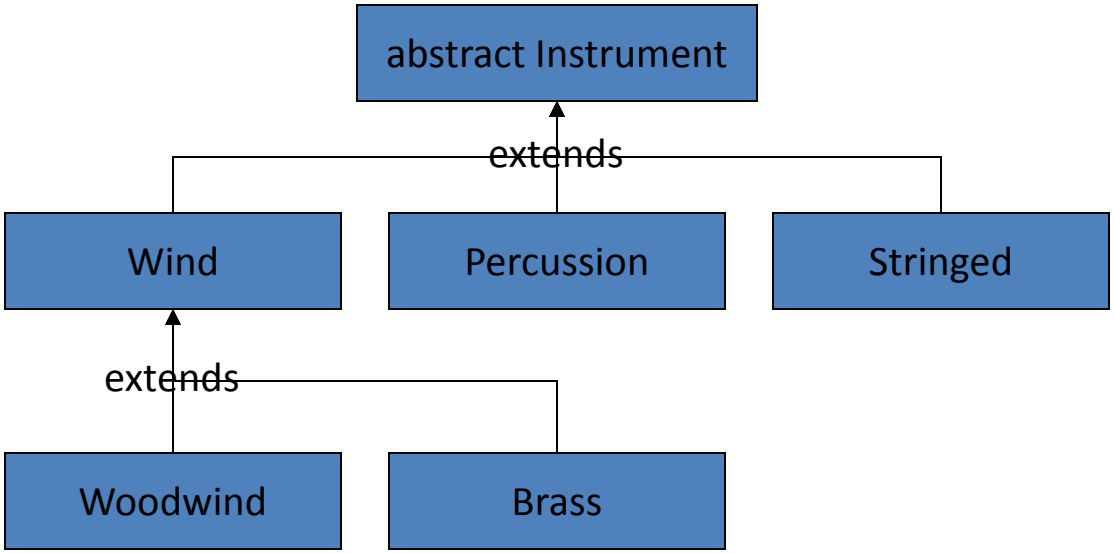
1. 接口的定义

- Java接口（Interface），是抽象方法与常量值的集合。
- 接口定义同类的定义类似，也是分为接口的声明和接口体，其中接口体由常量定义和方法定义两部分组成，基本格式如下：

```
[public] interface 接口名 [extends 父接口名列表]
{
    [public] [static] [final] 常量;
    [public] [abstract] 方法;
}
```

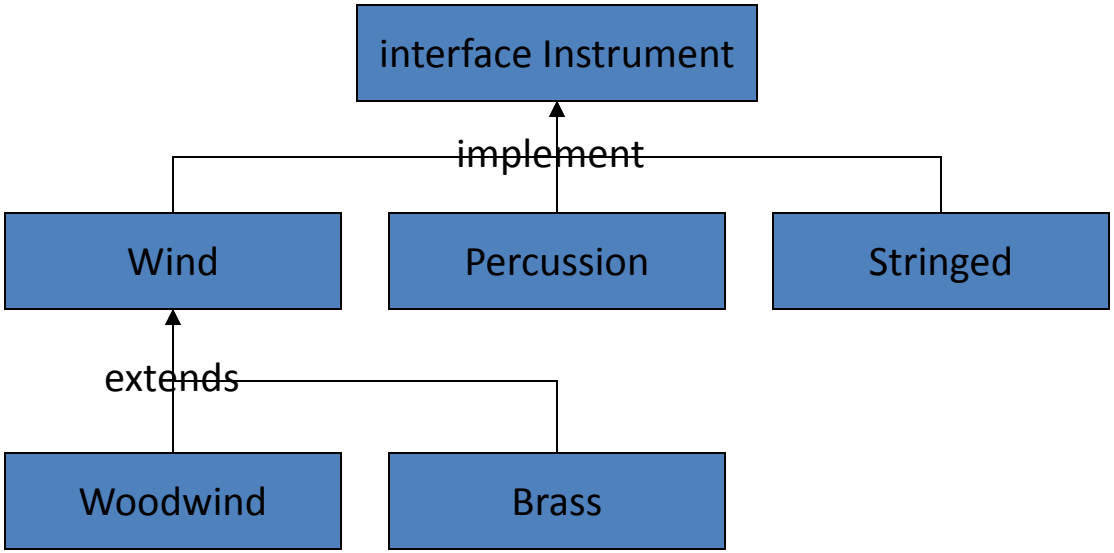
```
public interface Calculate {
   final float PI=3.14159f;                  //定义用于表示圆周率的常量PI
   float getArea(float r);                   //定义一个用于计算面积的方法
   float getCircumference(float r);  //定义一个用于计算周长的方法
}
```

4. 3接口




```
abstract class Instrument {
    public abstract void play();
    public String what() {return “Instrument”}
    public abstract void adjust();
}
```

4. 3接口

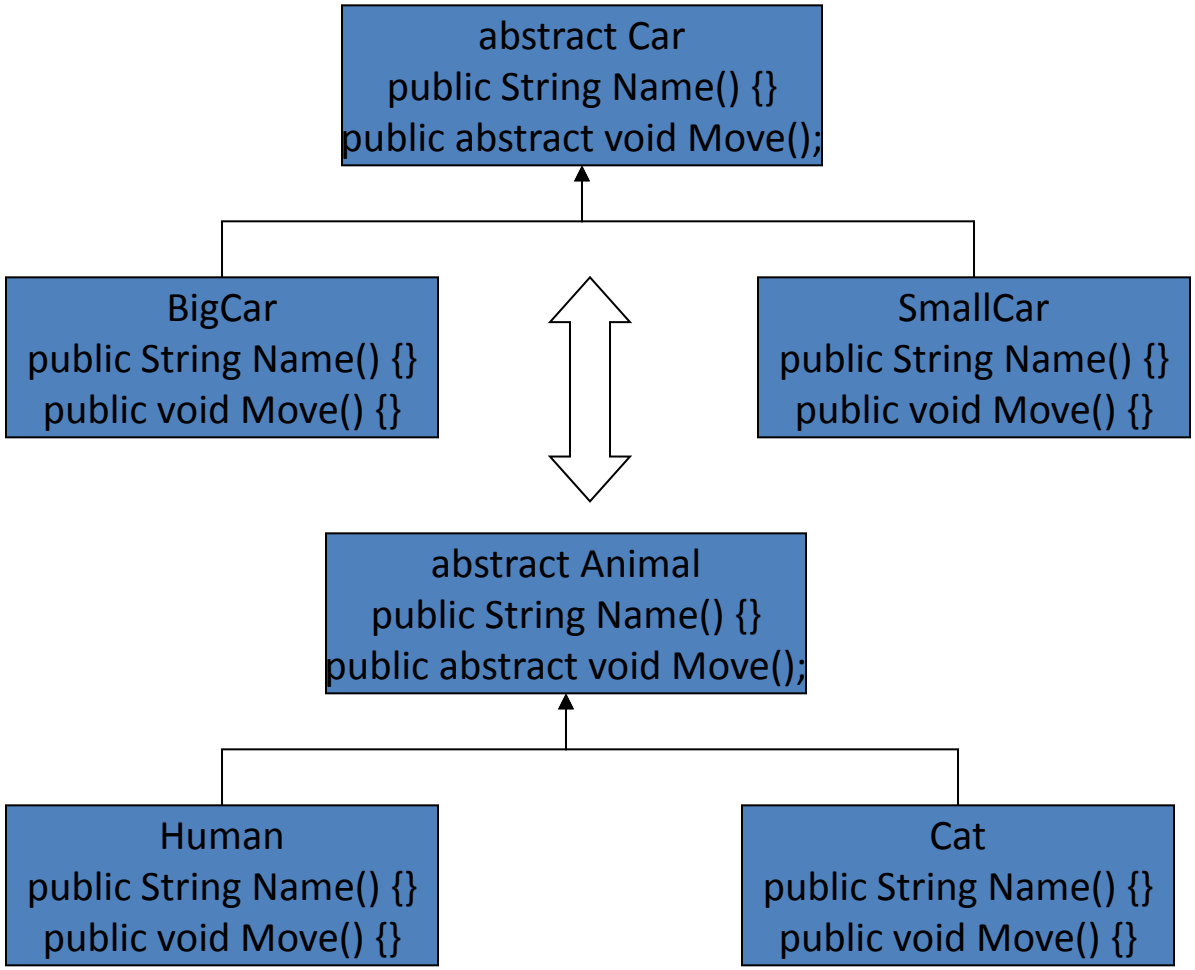



```
Interface Instrument {
    void play();
    String what();
    void adjust();
}
```

```
pubic class Wind implement Instrument {
    public void play(){System.out.println(“playing Wind”);}
    public String what() {return “Wind”;}
    public void adjust() {}
}
```

4. 3接口

- 抽象or接口？


abstract Car
public String Name() {}
public abstract void Move();
BigCar
public String Name() {}
public void Move() {}
SmallCar
public String Name() {}
public void Move() {}
abstract Animal
public String Name() {}
public abstract void Move();
Human
public String Name() {}
public void Move() {}
Cat
public String Name() {}
public void Move() {}

4. 3接口

- 选择接口

```
Interface HowToMove{
    void Move();
}
class Car implement HowToMove {
     public String Name(){…}
     public void Move() {…}
}
class Animal implement HowtoMove {
     public String Name(){…}
     public void Move() {…}
}
```

4.3 接口

2. Java接口的特征

- Java接口中的成员变量默认都是public,static,final类型的(都可省略),但必须被显示初始化,即接口中的成员变量为常量(大写,单词之间用“_”分隔)
- Java接口中的方法默认都是public,abstract类型的(都可省略),没有方法体,在接口中不能被实例化

```
public interface A
  {
   int CONST = 1;   //合法,CONST默认为public,static,final类型
   void method();  //合法,method()默认为public,abstract类型
   public abstract void method2();   //method2()显示声明为public,abstract类型
  }
```

4.3 接口

- Java接口中只能包含public,static,final类型的成员变量和public,abstract类型的成员方法

```
public interface A
 {
  int var;                          //错,var是常量,必须显示初始化
  void method(){...};               //错,接口中只能包含抽象方法
  protected void method2(); //错,接口中的方法必须是public类型

 }
```

- 接口中没有构造方法,不能被直接实例化

```
public interface A
 {
  public A(){...}; //错,接口中不能包含构造方法
  void method();
 }
```

4.3 接口

●一个接口不能实现(implements)另一个接口,但它可以继承多个其它的接口

```
public interface A
{   void methodA();
}
public interface B
{   void methodB();
}
public interface C extends A, B   //C称为复合接口
{   void methodC();
}
public interface C implements A{...}  //错
```

●一个类只能继承一个直接的超类,但可以实现多个接口,间接地实现了多重继承.

```
public class A extends B implements C, D{...}    //B为class,C,D为interface
```

4.3 接口

- Java接口必须通过类来实现它的抽象方法

```
public class A implements B{...}
```

- 当类实现了某个Java接口时,它必须实现接口中的所有抽象方法,否则这个类必须声明为抽象的
- 不允许创建接口的实例(实例化),但允许定义接口类型的引用变量,该引用变量引用实现了这个接口的类的实例

```
public class B implements A{}

A a = new B(); //引用变量a被定义为A接口类型,引用了B实例

A a = new A(); //错误,接口不允许实例化
```

3. 接口的应用

```
public interface Flyable {
   void fly();
}

public interface Talkable {
   void talk();
}

public interface Message {
   int MAX_SIZE = 4096;
   String getMessage();
}
```

```
class Parrot implements Flyable, Talkable {
    public void fly() {
       System.out.println("Flying like a parrot...");
     }
    public void talk() {
       System.out.println(“Hello! I am a parrot!”);
     }
}

class TextMessage implements Message {
    String message;
    public void setMessage(String msg) {
        message = msg;
        if (message.length() > MAX_SIZE)
            message = message.substring(0, MAX_SIZE);
    }
    public String getMessage() {
        return message;
    }
}
```

4.3 接口

•因为接口内的字段都是static和final的，所以我们可以很方便的利用这一点来创建一些常量。

```
public interface Constants {
    String ROOT = "/root";
    int MAX_COUNT = 200;
    int MIN_COUNT = 100;
}
```

在使用时可以直接用Constants.ROOT这样的形式来引用其中的常量。

•我们还可以用下面这种方法来创建初始值不确定的常量。

```
public interface RandomColor {
    int red = Math.random() * 255;
    int green = Math.random() * 255;
    int blue = Math.random() * 255;
}
```

其中red、green和blue的值会在第一次被访问时建立，然后保持不变。

3. 接口与抽象类的对比

➢**相同点：**

- 代表系统的抽象层,当一个系统使用一棵继承树上的类时,应该尽量把引用变量声明为继承树的上层抽象类型,这样可以提高两个系统之间的松耦合
- 都不能被实例化
- 都包含抽象方法,这些抽象方法用于描述系统能提供哪些服务,但不包含方法体

4.3 接口

3. 接口与抽象类的对比

➢不同点：

•最大的一个区别，就在于Java抽象类可以提供某些方法的部分实现，而Java接口不可以。可以向抽象类里加入一个新的具体方法，所有的子类都自动得到这个方法，但Java接口里加入一个新方法，必需手动给每个实现了该接口的类加上该方法的实现。

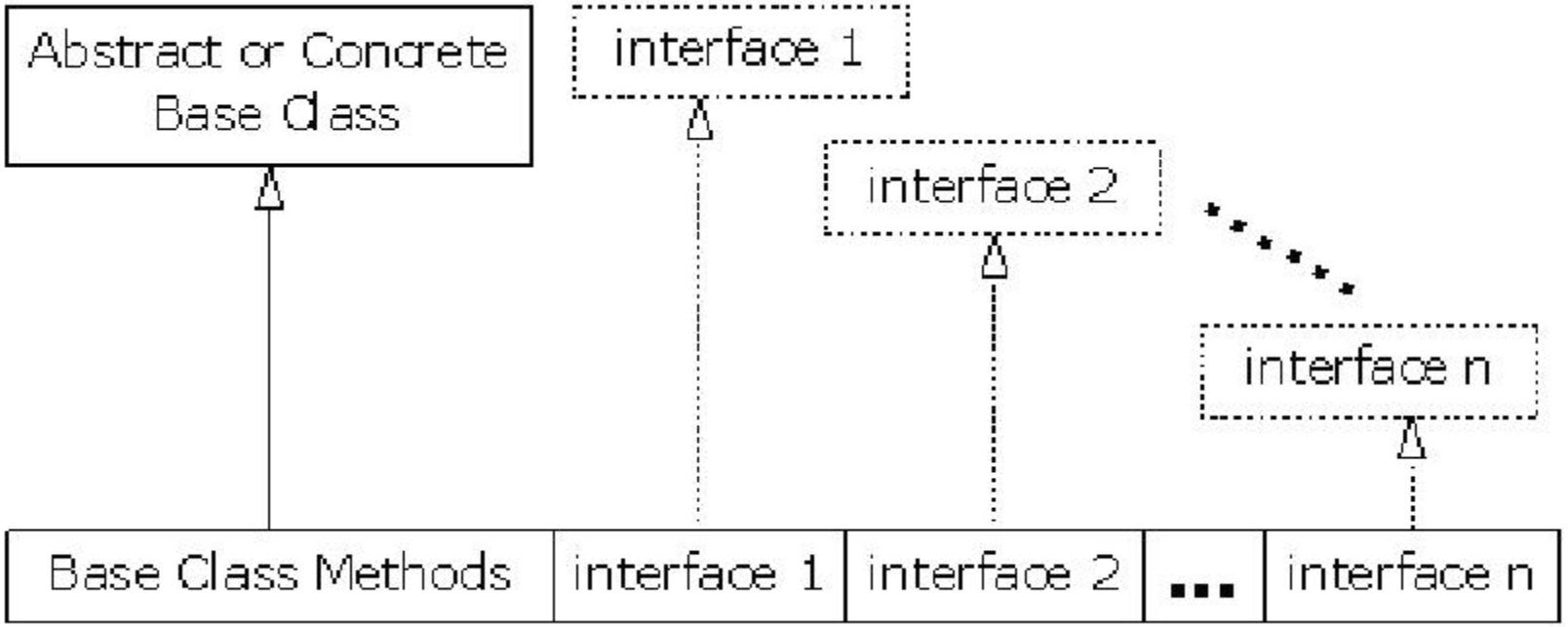
•抽象类的实现只能由子类给出，也即该实现只能在抽象类定义的继承的等级结构中。但是任何一个实现了一个Java接口所规定的方法的类都可以具有这个接口的类型，而一个类可以实现任意多个Java接口，从而这个类就有了多种类型。

4.3 接口

使用接口和抽象类的总体原则:

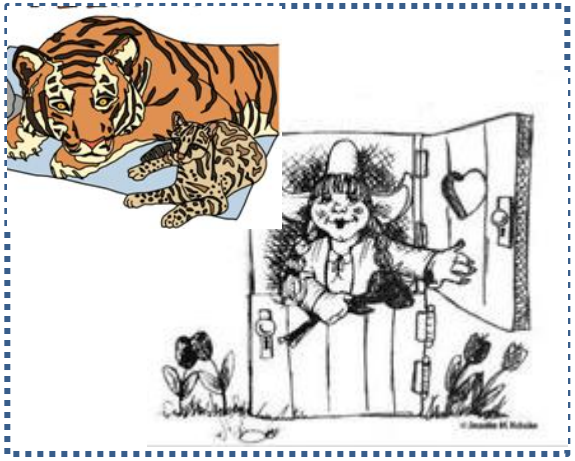
- 声明类型的工作仍然由Java接口承担，但是同时给出一个Java 抽象类，且实现了这个接口。
- 用接口作为系统与外界交互的窗口。
- Java接口本身必须非常稳定,Java接口一旦制定,就不允许随便更改,否则对外面使用者及系统本身造成影响。
- 用抽象类来定制系统中的扩展点，抽象类来完成部分实现,还要一些功能通过它的子类来实现。

4.3 接口


Abstract or Concrete Base Class
interface 1
interface 2
interface n
Base Class Methods
interface 1
interface 2
...
interface n

Fahter()
fun()
fun2()
method()

Father
类型对象



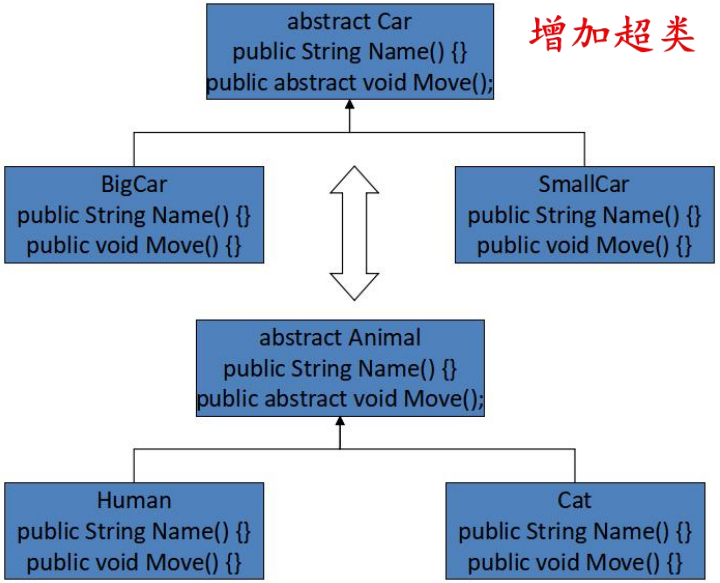
向上转型

Fahter
类型引用

Father → Son
类型对象

Fahter sample = **new** Son();

Son()
method2()



接口

4.3 接口

- 接口与工厂
  - 接口是实现多重继承的途径，而生成遵循某个接口的对象的典型方式就是工厂设计模式。
  - 在工厂上调用的是创建方法，不是构造方法，该工厂对象将生成某个接口实现的对象。

4.3 接口

```
interface Service {
  void method1();
  void method2();
}

interface ServiceFactory {
  Service getService();
}
class Implementation1 implements Service {
  Implementation1() {} // Package access
  public void method1() {print("Implementation1 method1");}
  public void method2() {print("Implementation1 method2");}
}

class Implementation1Factory implements ServiceFactory {
  public Service getService() {
    return new Implementation1();
  }
}
```

4.3 接口

```
class Implementation2 implements Service {
  Implementation2() {} // Package access
  public void method1() {print("Implementation2 method1");}
  public void method2() {print("Implementation2 method2");}
}

class Implementation2Factory implements ServiceFactory {
  public Service getService() {
    return new Implementation2();
  }
}
public class Factories {
  public static void serviceConsumer(ServiceFactory fact) {
    Service s = fact.getService();
    s.method1();
    s.method2();

  }
  public static void main(String[] args) {
    serviceConsumer(new Implementation1Factory());
    // Implementations are completely interchangeable:
    serviceConsumer(new Implementation2Factory());
  }
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:~
```

4.3 接口

- 接口与工厂
  - 接口是实现多重继承的途径，而生成遵循某个接口的对象的典型方式就是工厂设计模式。
  - 在工厂上调用的是创建方法，不是构造方法，该工厂对象将生成某个接口实现的对象。
  - 通过这种方式，我们的代码将完全与接口的实现分离。

内容提要

- 4.1 类的复用
- 4.2 多态
- 4.3 接口
- 4.4 内部类

4.4 内部类Inner Classes

1. 内部类的定义
2. 内部类的分类和作用
3. 内部类与外部类

内部类从形式上，对类继承和接口的使用提供了更为隐蔽和灵活的方式。

4. 4内部类

1. 内部类的定义和特性

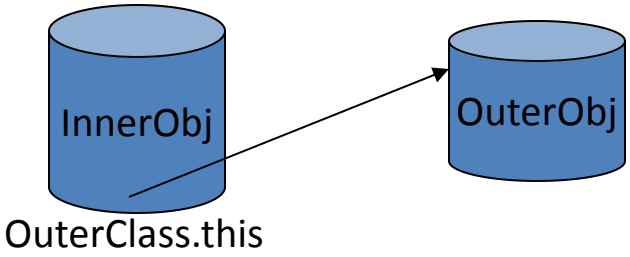
- 内部类，一个类的定义放在另一个类的内部，这个放在内部的类就叫做内部类。它有两种形式：
  - 嵌套类(nested class)：静态的，使用很少；
  - 内部类(inner class)：非静态，较重要。
- 内部类可以像方法一样被修饰为public，default，protected和private，也可以是静态static的，也就是嵌套类。

```
public class First {
  class  Contents{
     public void f(){
        System.out.println("In Class First's inner Class Contents method f()");
     }
  }
}
```

```
interface Contents {
    int value();
}
 interface Destination {
    String readLabel();
}
class Goods {
    private class Content implements Contents {
        private int i = 11;

        public int value() {
            return i;
        }
    }
    protected class GDestination implements Destination {
        private String label;

        private GDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() {
            return label;
        }
    }
    public Destination dest(String s) {
        return new GDestination(s);
    }
    public Contents cont() {
        return new Content();
    }
}
public class TestGoods {
    public static void main(String[] args)
        Goods p = new Goods();
        Contents c = p.cont();
        Destination d = p.dest("Beijing");
    }
}
```

在这个例子里类Content和GDestination被定义在了类Goods内部，并且分别有着protected和private修饰符来控制访问级别。Content代表着Goods的内容，而GDestination代表着Goods的目的地。它们分别实现了两个接口Content和Destination。在后面的main方法里，直接用 Contents c和Destination d进行操作，你甚至连这两个内部类的名字都没有看见。这样，内部类的第一个好处就体现出来了隐藏你不想让别人知道的操作，也即封装性。

4. 4内部类



●可以这样创建内部类的对象：

```
outerClass outerObject=new outerClass(Constructor Parameters);
outerClass.innerClass innerObject=
        outerObject.new InnerClass(Constructor Parameters);
```

注意在创建非静态内部类对象时，一定要先创建起相应的外部类对象。

4.4内部类

2. 内部类分类和作用

➢静态内部类（嵌套类）：

将内部类声明为static，这通常称为嵌套类（nested class）。

特性：

•要创建嵌套类的对象，并不需要其外部类的对象。

outerClass.innerClass innerObject=

new **outerClass**.innerClass(Constructor Parameters);

•不能从嵌套类的对象中访问非静态的外部类成员。

•非嵌套类中，不能有静态数据、静态方法、静态内部类

4.4内部类

2. 内部类分类和作用

➢静态内部类（嵌套类）：

```
public interface ClassInInterface {
    void hello();
    class Test implements ClassInInterface{
        public void hello(){
            System.out.println("hello");
        }
    }

}
```

```
public class Outer {
    private static int i =1;
    private int j = 10;
    public static void outer_f1(){
    }
    public void outer_f2(){
    }
    static class Inner{
        static int inner_i =100;
        int inner_j = 200;
        static void inner_f1(){
            System.out.println("Outer.i"+i);
            outer_f1();
        }
        void inner_f2(){
            System.out.println("Outer.j"+j);
            outer_f2();
        }
    }
    public void outer_f3(){
        System.out.println(Inner.inner_i);
        Inner.inner_f1();
        Inner inner = new Inner();
        inner.inner_f2();
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        new Outer().outer_f3();
    }

}
```

2. 内部类分类和作用

➢局部内部类：

Java内部类也可以是局部的，它可以定义在一个方法甚至一个代码块之内。

- 不能有访问控制符
- 访问局部 内部类必须有外部类对象；
- 局部内部类可以访问外部类的成员变量；
- 但局部内部类只能访问外部类的final类型局部变量。

➢局部内部类：

Java内部类也可以是局部的，它可以定义在一个方法甚至一个代码块之内。

```
public class Outer1 {
    private int s = 100;
    private int out_i = 1;
    public void f(final int k) {
        final int s = 200;
        int i = 1;
        final int j = 10;
        // 定义在方法内部
        class Inner {
            int s = 300;// 可以定义与外部类同名的变量
            // static int m = 20;//不可以定义静态变量
            Inner(int k) {
                inner_f(k);
            }
            int inner_i = 100;
            void inner_f(int k) {
                // 如果内部类没有与外部类同名的变量，在内部类中可以直接访问外部类的实例变量
                System.out.println(out_i);
                // 可以访问外部类的局部变量(即方法内的变量)，但是变量必须是final的
                System.out.println(j);
                // System.out.println(i);
                // 如果内部类中有与外部类同名的变量，直接用变量名访问的是内部类的变量
                System.out.println(s);
                // 用this.变量名访问的也是内部类变量
                System.out.println(this.s);
                // 用外部类名.this.内部类变量名访问的是外部类变量
                System.out.println(Outer1.this.s);
            }
        }
        new Inner(k);
    }
    public static void main(String[] args) {
        // 访问局部内部类必须先有外部类对象
        Outer1 out = new Outer1();
        out.f(3);
    }
}
```

1 2 3 4 5 6 7 8

```
1
10
300
300
100
```

4.4内部类

➢匿名内部类：

当你只需要创建一个类的对象而且用不上它的名字时，使用匿名内部类可以使代码看上去简洁清楚。它的语法规则是这样的：

new interfacename(){.....};

或 new superclassname(){.....};

```
public class Goods3 {
   public Contents cont() {
      return new Contents() {
         private int i = 11;
         public int value() {
            return i;
         }
      };
   }
}
```

```
interface Contents{
   int value();
}
```

实现了Contents接口的实例类，但是并没有显示的给出名字及对象名称

4.4内部类

➢匿名内部类：

当你只需要创建一个类的对象而且用不上它的名字时，使用匿名内部类可以使代码看上去简洁清楚。它的语法规则是这样的：

new interfacename(){.....};

或 new superclassname(){.....};

```
public class Goods3 {
   public Contents cont() {
      return new Contents() {
         private int i = 11;
         public int value() {
            return i;
         }
      };
   }
}
```

```
interface Contents {
    int value();
}
 interface Destination {
    String readLabel();
}
class Goods {
    private class Content implements Contents {
        private int i = 11;

        public int value() {
            return i;
        }
    }

    public Contents cont() {
        return new Content();
    }
```

4.4内部类

- 匿名内部类

```
public class Parcel7 {
  public Contents contents() {
    return new Contents() {
      private int i = 11;
      public int value() { return i; }
    };
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Contents c = p.contents();
  }
} ///:~
public class Parcel7b {
  class MyContents implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  public Contents contents() { return new MyContents(); }
  public static void main(String[] args) {
    Parcel7b p = new Parcel7b();
    Contents c = p.contents();
  }
} ///:~
```

4.4内部类

- 匿名内部类

```
interface Service {
  void method1();
  void method2();
}

interface ServiceFactory {
  Service getService();
}

class Implementation1 implements Service {
  private Implementation1() {}
  public void method1() {print("Implementation1 method1");}
  public void method2() {print("Implementation1 method2");}
  public static ServiceFactory factory =
    new ServiceFactory() {
      public Service getService() {
        return new Implementation1();
      }
    };
}
```

4.4内部类

- 匿名内部类

```
class Implementation2 implements Service {
  private Implementation2() {}
  public void method1() {print("Implementation2 method1");}
  public void method2() {print("Implementation2 method2");}
  public static ServiceFactory factory =
    new ServiceFactory() {
      public Service getService() {
        return new Implementation2();
      }
    };
}

public class Factories {
  public static void serviceConsumer(ServiceFactory fact) {
    Service s = fact.getService();
    s.method1();
    s.method2();
  }
  public static void main(String[] args) {
    serviceConsumer(Implementation1.factory);
    // Implementations are completely interchangeable:
    serviceConsumer(Implementation2.factory);
  }
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:~
```

4. 4内部类

在java的事件处理的匿名适配器中，匿名内部类被大量的使用。例如在想关闭窗口时加上这样一句代码：

```
frame.addWindowListener(new WindowAdapter(){
   public void windowClosing(WindowEvent e){
     System.exit(0);
   }
});
```

有一点需要注意的是，匿名内部类由于没有名字，所以它没有构造函数（但是如果这个匿名内部类继承了一个只含有带参数构造函数的超类，创建它的时候必须带上这些参数，并在实现的过程中使用super关键字调用相应的内容）。

4. 4内部类

➢内部类存在的作用

●一般的非内部类，是不允许有 private 与protected 权限的，但内部类可以。内部类作用是更小层次的封装，把一个类隐藏在另一个类的内部，只让它的外部类看得到它，能更方便的在内部类中访问外部类的私有成员。

●由于内部类对外部类的所有内容都是可访问的，我们就可以考虑使用内部类来实现接口。这样一来，内部类实现了对主类方法的封装，不影响接口的扩展，可以避免修改接口而实现同一个类中两种同名方法的调用。

```
public interface A{
    void f();
    void p();}
```

```
public interface B{
    void f();
    void q();}
```

```
public class C implement A,B {
    内部类分别封装
}
```

4. 4内部类

3. 内部类与外部类

➢内部类生成的class文件

●内部类与外部类在编译时各自生成.class文件，内部类的名字：OutClassName$InnerClassName.class

●匿名内部类为：OutClassName$#.class，#为数字，从1开始。

3. 内部类与外部类

➢内部类访问外部类

●内部类对象与创造它的外部类对象存在着固有联系，一旦内部类创建，就可以在不需任何特殊条件的情况下访问外部类的所有成员。

```
public class First {
  class Contents{
    public void getStr(){
        System.out.println("First.str="+str);
    }
  }
  private String str;
}
```

4.4内部类

●使用关键字.this与.new

我们要得到对外部类的引用，可以在内部类中使用.this关键字。

```
public class Outer {
    private int num;

    public Outer() {
    }

    public Outer(int num) {
        this.num = num;
    }

    private class Inner {
        public Outer getOuter() {
            return Outer.this;
        }

        public Outer newOuter() {
            return new Outer();
        }
    }

    public static void main(String[] args) {
        Outer test = new Outer(5);
        Outer.Inner inner = test.new Inner();
        Outer test2 = inner.getOuter();
        Outer test3 = inner.newOuter();
        System.out.println(test2.num);
        System.out.println(test3.num);
    }
}
```

5
0

我们要仔细注意this和new在这里的不同，内部类Inner内方法getOuter()使用.this得到是创建该内部类时使用的外部类对象的引用，而newOuter()方法使用new创建了一个新的引用。

OutClass.InnerClass obj = outClassInstance.new InnerClass();

4.4内部类

- 内部类与向上转型

  将内部类向上转型为超类型，尤其是接口时，内部类就有了用武之地。

```
interface Shape {
    public void paint();
}

public class Painter {

    private class InnerShape implements Shape {
        public void paint() {
            System.out.println("painter paint() method");
        }
    }

    public Shape getShape() {
        return new InnerShape();
    }

    public static void main(String[] args) {
        Painter painter = new Painter();
        Shape shape = painter.getShape();
        shape.paint();
    }
}
```

```
painter paint() method
```

4.4内部类

➢内部类的重写问题

```
class Egg {
    private Yolk y;

    protected class Yolk {
        public Yolk() {
            System.out.println("Egg.Yolk()");
        }
    }

    public Egg() {
        System.out.println("New Egg()");
        y = new Yolk();
    }
}

public class BigEgg extends Egg {
    public class Yolk {
        public Yolk() {
            System.out.println("BigEgg.Yolk()");
        }
    }

    public static void main(String[] args) {
        new BigEgg();
    }
}
```

缺省的构造方法是编译器自动生成的，这里是调用超类的缺省构造方法。重写的两个内部类是完全独立的两个实体，各自在自己的命名空间内。

```
New Egg()
Egg.Yolk()
```

```
class Egg2 {
    protected class Yolk {
        public Yolk() {
            System.out.println("Egg2.Yolk()");
        }
        public void f() {
            System.out.println("Egg2.Yolk.f()");
        }
    }
    private Yolk y = new Yolk();
    public Egg2() {
        System.out.println("New Egg2()");
    }
    public void insertYolk(Yolk yy) {
        y = yy;
    }
    public void g() {
        y.f();
    }
}
public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() {
            System.out.println("BigEgg2.Yolk()");
        }
        public void f() {
            System.out.println("BigEgg2.Yolk.f()");
        }
    }
    public BigEgg2() {
        insertYolk(new Yolk());
    }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
}
```

Egg2$Yolk

BigEgg2$Yolk

4 8 3 5 11 13 2 7 9 14 6 10 1 12

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

4.4内部类

➢内部类的继承问题

```
class WithInner {
    class Inner {
        Inner() {
            System.out.println("this is a constructor in WithInner.Inner");
        };
    }
}

public class InheritInner extends WithInner.Inner {
    // ! InheritInner() {} // Won't compile
    InheritInner(WithInner wi) {
        wi.super();
        System.out.println("this is a constructor in InheritInner");
    }

    public static void main(String[] args) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
}
```

this is a constructor in WithInner.Inner
this is a constructor in InheritInner

那个“秘密的”外部类对象的引用必须被初始化，而在被继承的类中并不存在要联接的缺省对象。要解决这个问题，需使用专门的语法来明确说清它们之间的关联。

Enjoy your