

接口

reference

- <http://bobah.net/book/export/html/55>
- [http://stackoverflow.com/questions/1504633/what-is-the-point-of-
invokeinterface](http://stackoverflow.com/questions/1504633/what-is-the-point-of-
invokeinterface)
- [http://www.javaworld.com/article/2073649/core-java/why-extends-is-
evil.html](http://www.javaworld.com/article/2073649/core-java/why-extends-is-
evil.html)
- [http://www.javaworld.com/article/2076814/core-java/inheritance-
versus-composition--which-one-should-you-choose-.html](http://www.javaworld.com/article/2076814/core-java/inheritance-
versus-composition--which-one-should-you-choose-.html)
- <http://ebnbin.com/2015/12/20/java-8-default-methods/>

Outline

- 接口
- 类 vs 接口
- invokevirtual vs invokeinterface
- default method in Java 8

接口

案例

- 设计一个宠物商店，包含宠物狗。
- 宠物需要具有 beFriendly() 和 Play() 方法。
- 宠物商店程序会有其它宠物，共享行为模型，但具体怎么做不同。

So what we **REALLY** need is:

- ✱ A way to have pet behavior in just the pet classes
- ✱ A way to guarantee that all pet classes have all of the same methods defined (same name, same arguments, same return types, no missing methods, etc.), without having to cross your fingers and hope all the programmers get it right.
- ✱ A way to take advantage of polymorphism so that all pets can have their pet methods called, without having to use arguments, return types, and arrays for each and every pet class.

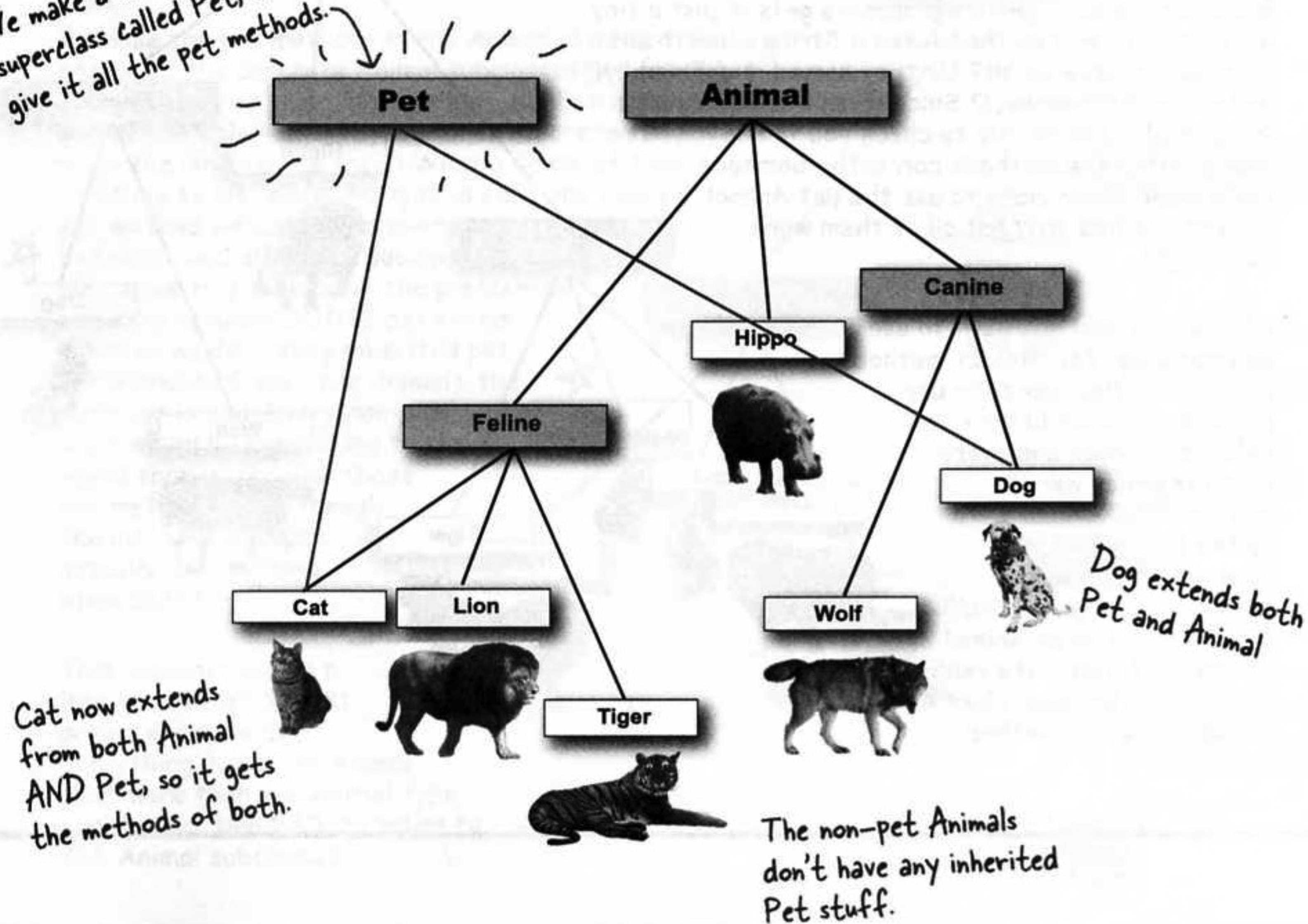
在宠物类中定义宠物行为

宠物共享行为

多态实现动态绑定

It looks like we need **TWO** superclasses at the top

We make a new abstract superclass called Pet, and give it all the pet methods.



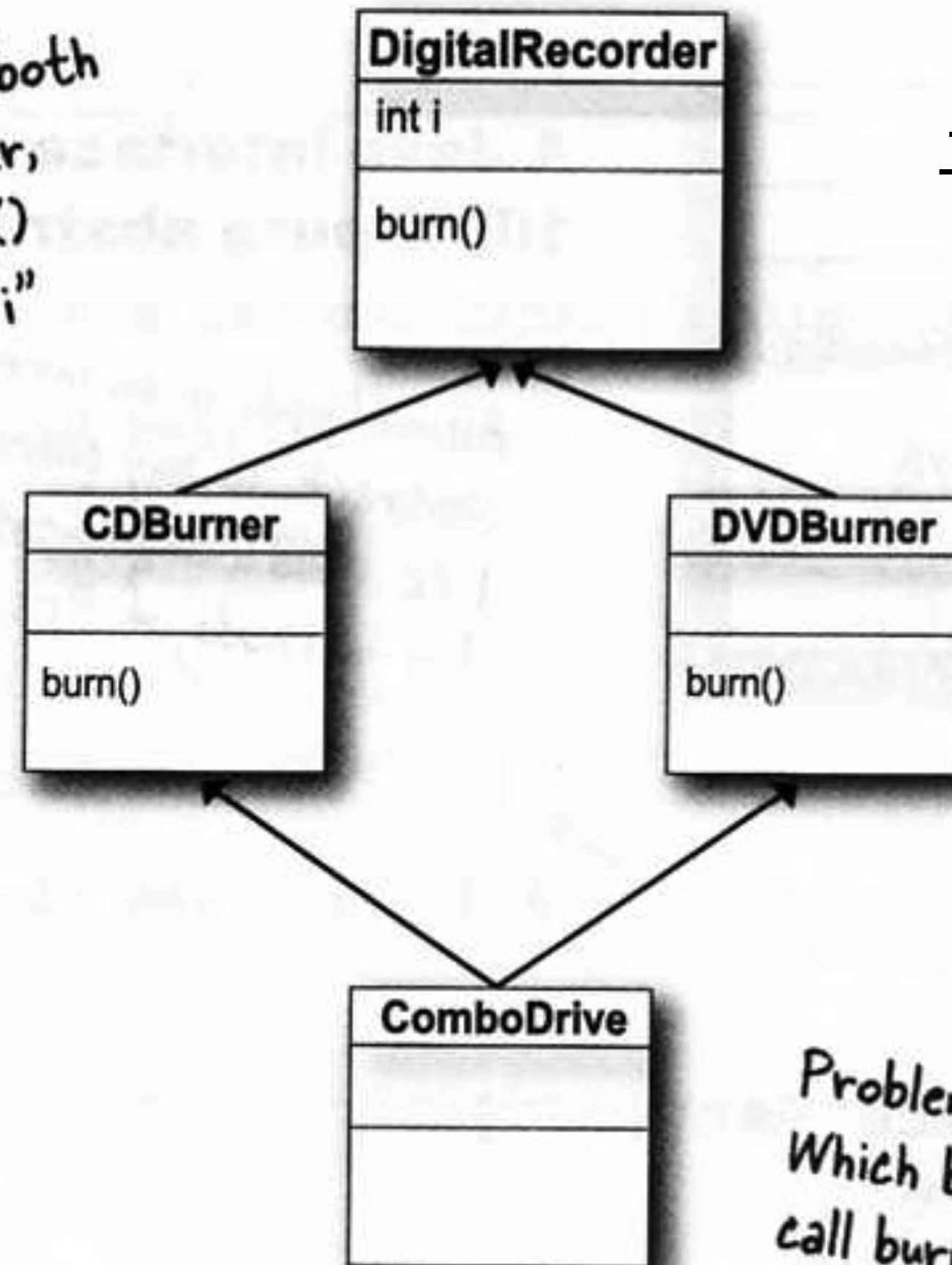
It's called "multiple inheritance" and it can be a Really Bad Thing.

That is, if it were possible to do in Java.

But it isn't, because multiple inheritance has a problem
known as The Deadly Diamond of Death.

Deadly Diamond of Death

CDBurner and DVDBurner both
inherit from DigitalRecorder,
and both override the burn()
method. Both inherit the "i"
instance variable.



子类既继承了 `i`，也重写了 `burn` 方法

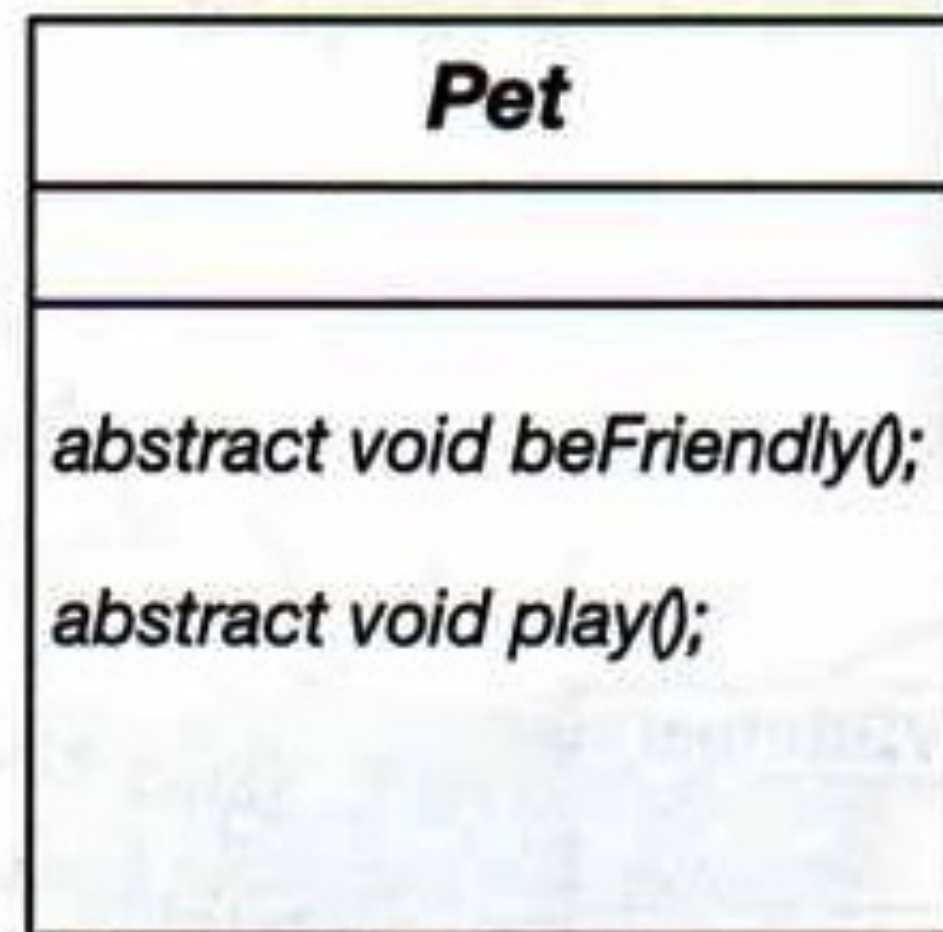
Imagine that the "i" instance
variable is used by both CDBurner
and DVDBurner, with different
values. What happens if ComboDrive
needs to use both values of "i"?

Problem with multiple inheritance.
Which burn() method runs when you
call burn() on the ComboDrive?

Interface

A Java interface solves your multiple inheritance problem by giving you much of the polymorphic *benefits* of multiple inheritance without the pain and suffering from the Deadly Diamond of Death (DDD).

The way in which interfaces side-step the DDD is surprisingly simple: *make all the methods abstract!* That way, the subclass *must* implement the methods (remember, abstract methods *must* be implemented by the first concrete subclass), so at runtime the JVM isn't confused about *which* of the two inherited versions it's supposed to call.



**A Java interface is like a
100% pure abstract class.**

All methods in an interface are abstract, so any class that IS-A Pet **MUST** implement (i.e. override) the methods of Pet.

To DEFINE an interface:

```
public interface Pet {...}
```

Use the keyword "interface"
instead of "class"

To IMPLEMENT an interface:

```
public class Dog extends Canine implements Pet {...}
```

Use the keyword "implements" followed
by the interface name. Note that
when you implement an interface you
still get to extend a class

Making and Implementing the Pet interface

You say 'interface' instead of 'class' here

```
public interface Pet {
```

```
    public abstract void beFriendly();
```

```
    public abstract void play();
```

```
}
```

interface methods are implicitly public and abstract, so typing in 'public' and 'abstract' is optional (in fact, it's not considered 'good style' to type the words in, but we did here just to reinforce it, and because we've never been slaves to fashion...)

All interface methods are abstract, so they **MUST** end in semicolons. Remember, they have no body!

Dog IS-A Animal
and Dog IS-A Pet

```
public class Dog extends Canine implements Pet {
```

```
    public void beFriendly() {...}
```

```
    public void play() {...}
```

```
    public void roam() {...}
```

```
    public void eat() {...}
```

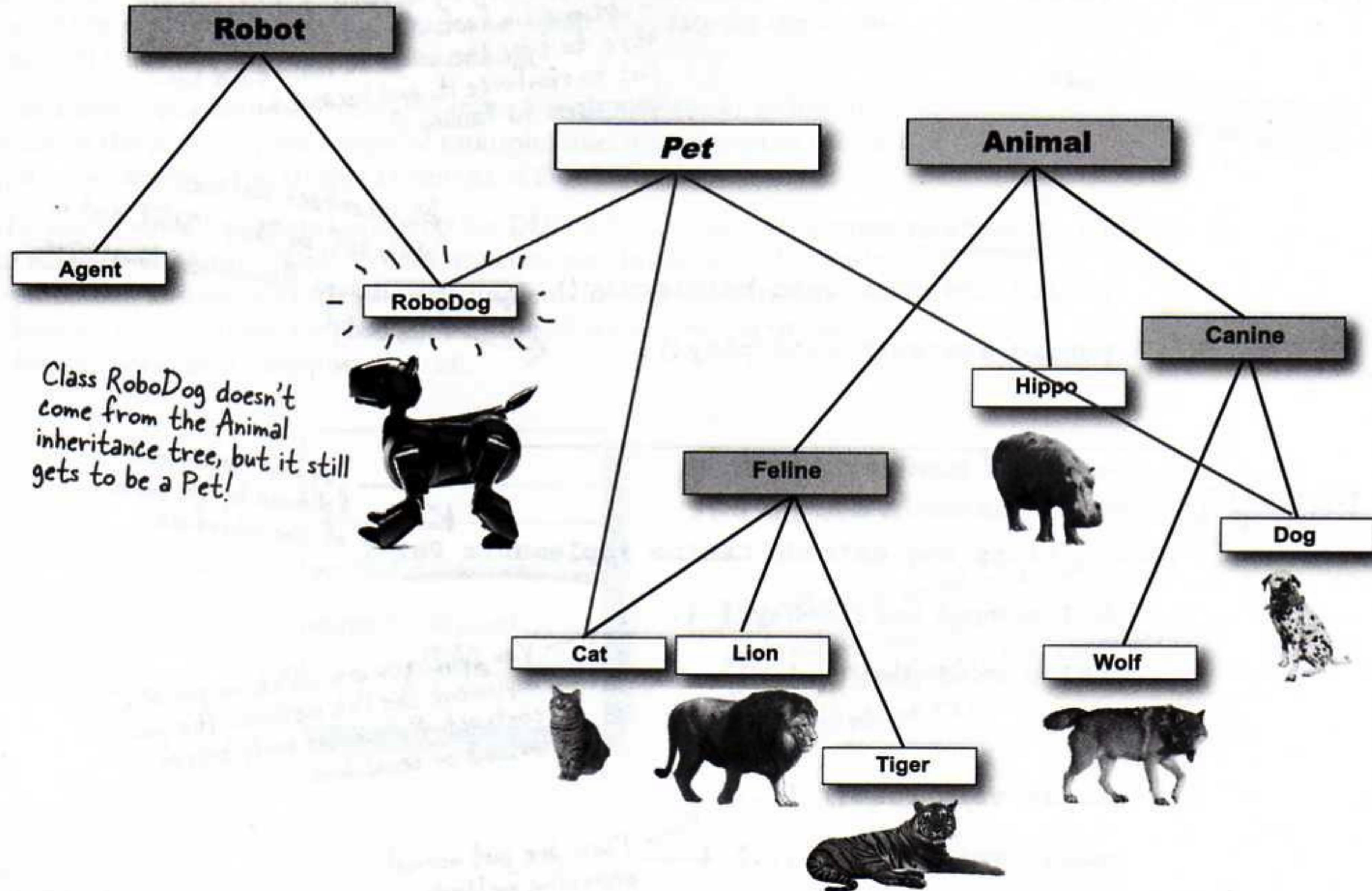
```
}
```

You say 'implements' followed by the name of the interface.

You SAID you are a Pet, so you **MUST** implement the Pet methods. It's your contract. Notice the curly braces instead of semicolons.

These are just normal overriding methods.

Classes from *different* inheritance trees can implement the *same* interface.



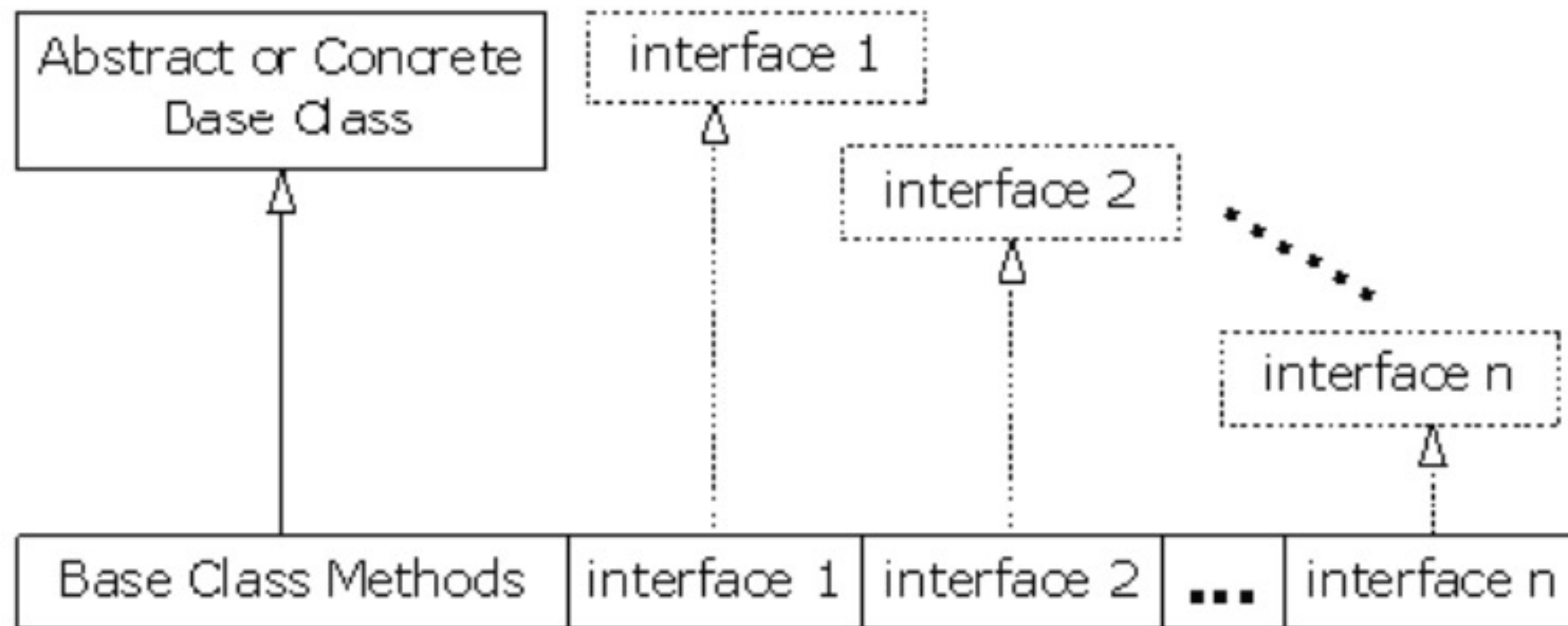
Extends one & Implements more

Better still, a class can implement *multiple* interfaces!

A Dog object IS-A Canine, and IS-A Animal, and IS-A Object, all through inheritance. But a Dog IS-A Pet through interface implementation, and the Dog might implement other interfaces as well. You could say:

```
public class Dog extends Animal implements  
Pet, Saveable, Paintable { ... }
```


- 实现规范和约束
- 实现灵活的多重继承
- 解耦和模块化



How do you know whether to make a class, a subclass, an abstract class, or an interface?

- Make a class that doesn't extend anything (other than Object) when your new class doesn't pass the IS-A test for any other type.
- Make a subclass (in other words, *extend* a class) only when you need to make a more specific version of a class and need to override or add new behaviors.
- Use an abstract class when you want to define a template for a group of subclasses, and you have at least *some* implementation code that all subclasses could use. Make the class abstract when you want to guarantee that nobody can make objects of that type.
- Use an interface when you want to define a role that other classes can play, regardless of where those classes are in the inheritance tree.

非is-a关系
尽可能不要用继承

is-a关系
特殊实现用继承

一组类的共性模板用抽象

定义一个类的
功能性用接口

通过继承扩展接口

- `//: c08:HorrorShow.java`
- `// Extending an interface with inheritance.`
- `interface Monster {`
- `void menace();`
- `}`
- `interface DangerousMonster extends Monster {`
- `void destroy();`
- `}`

- `//: c08:RandVals.java`
- `// Initializing interface fields with`
- `// non-constant initializers.`
- `import java.util.*;`
- `public interface RandVals {`
 - `Random rand = new Random();`
 - `int randomInt = rand.nextInt(10);`
 - `long randomLong = rand.nextLong() * 10;`
 - `float randomFloat = rand.nextLong() * 10;`
 - `double randomDouble = rand.nextDouble() * 10;`
- `} ///:~`
- 在接口中定义的数据成员自动是**static** 和**final** 的。它们不能是“空final”，但是可以被非常量表达式初始化。
- 这些数据成员不是接口的一部分，只是被存储在该接口的静态存储区域内。

```

public interface InterfaceTest {

    int a = 10;//全局变量默认强制是 public static final

    public static void staticMethod() { //静态方法，JDK1.8下编译通过
        System.out.println("");
    }

    int abstractMethod(); //抽象方法

    public static abstract class InnerClass{ //抽象内部类，默认强制 public static
        //.....
    }

    enum MyEnum{RED,BLUE,GRREN} //枚举类，默认强制 public static

    interface InnerInterface{ //嵌套接口，默认强制 public static
        void aa();
    }
}

class MyClass implements InterfaceTest{ //实现上面的接口

    @Override
    public int abstractMethod() { //实现抽象方法

        return 0;
    }
}

```

接口可以嵌套在类或其它接口中


```
interface A{  
void sameMethodA();  
}
```

```
interface B{  
void sameMethodA();  
void sameMethodB(int a); //返回类型是void  
}
```

```
interface C{  
int sameMethodB(int a); //返回类型是 int  
}
```

```
interface D extends A,B{ //编译通过，即使A，B定义了相同方法  
}
```

```
interface E extends B,C{ //编译失败  
}
```

Invoking the superclass version of a method

```
abstract class Report {  
    void runReport() {  
        // set-up report  
    }  
    void printReport() {  
        // generic printing  
    }  
}
```

← superclass version of the method does important stuff that subclasses could use

```
class BuzzwordsReport extends Report {  
    void runReport() {  
        super.runReport();  
        buzzwordCompliance();  
        printReport();  
    }  
    void buzzwordCompliance() {...}  
}
```

← call superclass version, then come back and do some subclass-specific stuff

class vs interface

具体类和接口的使用

- 显示地使用具体的类
 - 锁定某个具体的实现
 - 丧失了可扩展性
 - 丧失了灵活性
- 按接口编程
 - 增加开发的可并行性

案例 1：遍历集合

Design I - 使用具体类

```
f()
{
    LinkedList list = new LinkedList();
    //...
    g( list );
}

g( LinkedList list )
{
    list.add( ... );
    g2( list )
}
```

如果我们有这样的需求变更：
我们希望快速的查找

Design II – 面向对象、父类

```
f()
{
    Collection list = new LinkedList();
    //...
    g( list );
}

g( Collection list )
{
    list.add( ... );
    g2( list )
}
```

```
f()
{
    Collection c = new HashSet();
    //...
    g( c );
}

g( Collection c )
{
    for( Iterator i = c.iterator(); i.hasNext() ;)
        do_something_with( i.next() );
}
```

继承的问题

- 子类必须保持和父类同样的接口
- 子类必须继承了父类的实现
- 一旦父类脆弱发生变化，就会对子类造成很大的麻烦

集合的问题

- 使用集合给的比要的更多
- 增加了耦合性
- 增加了出错的概率

```
f()
{
    Collection c = new HashSet();
    //...
    g( c );
}

g( Collection c )
{
    for( Iterator i = c.iterator(); i.hasNext() ;)
        do_something_with( i.next() );
}
```


As another example, compare this code:

```
f()
{
    Collection c = new HashSet();
    //...
    g( c );
}
g( Collection c )
{
    for( Iterator i = c.iterator(); i.hasNext() ;)
        do_something_with( i.next() );
}
```

按接口编程

to this:

```
f2()
{
    Collection c = new HashSet();
    //...
    g2( c.iterator() );
}
g2( Iterator i )
{
    while( i.hasNext() ;)
        do_something_with( i.next() );
}
```

Design III — 使用接口

案例 2：记录最高水位的栈

Design I

```
class Stack extends ArrayList
{
    private int stack_pointer = 0;
    public void push( Object article )
    {
        add( stack_pointer++, article );
    }
    public Object pop()
    {
        return remove( --stack_pointer );
    }
    public void push_many( Object[] articles )
    {
        for( int i = 0; i < articles.length; ++i )
            push( articles[i] );
    }
}
```

```
Stack a_stack = new Stack();
a_stack.push("1");
a_stack.push("2");
a_stack.clear();
```

Disadvantage

- 首先，如果重写所有方法，那么父类应该是一个接口（定义规范），而不是一个类。即不使用任何继承的方法，实现继承毫无意义。
- 其次，我们不希望一个栈支持所有ArrayList的方法

Design II

```
class Stack
{
    private int stack_pointer = 0;
    private ArrayList the_data = new ArrayList();
    public void push( Object article )
    {
        the_data.add( stack_pointer++, article );
    }
    public Object pop()
    {
        return the_data.remove( --stack_pointer );
    }
    public void push_many( Object[] articles )
    {
        for( int i = 0; i < o.length; ++i )
            push( articles[i] );
    }
}
```

Design III

```
class Monitorable_stack extends Stack
{
    private int high_water_mark = 0;
    private int current_size;
    public void push( Object article )
    {   if( ++current_size > high_water_mark )
        high_water_mark = current_size;
        super.push(article);
    }

    public Object pop()
    {   --current_size;
        return super.pop();
    }

    public int maximum_size_so_far()
    {   return high_water_mark;
    }
}
```

在Stack上创建一个变量，
用于监测栈的大小

- This new class works well, at least for a while. Unfortunately, the code exploits the fact that `push_many()` does its work by calling `push()`. At first, this detail doesn't seem like a bad choice. It simplifies the code, and you get the derived class version of `push()`, even when the `Monitorable_stack` is accessed through a `Stack` reference, so the `high_water_mark` updates correctly.

- One fine day, someone might run a profiler and notice the Stack isn't as fast as it could be and is heavily used. You can rewrite the Stack so it doesn't use an ArrayList and consequently improve the Stack's performance.

Design IV

```
class Monitorable_stack extends Stack
{
    private int high_water_mark = 0;
    private int current_size;
    public void push( Object article )
    {    if( ++current_size > high_water_mark )
        high_water_mark = current_size;
        super.push(article);
    }

    public Object pop()
    {    --current_size;
        return super.pop();
    }

    public int maximum_size_so_far()
    {    return high_water_mark;
    }
}
```

```
class Stack
{    private int stack_pointer = -1;
    private Object[] stack = new Object[1000];
    public void push( Object article )
    {    assert stack_pointer < stack.length;
        stack[ ++stack_pointer ] = article;
    }
    public Object pop()
    {    assert stack_pointer >= 0;
        return stack[ stack_pointer-- ];
    }
    public void push_many( Object[] articles )
    {    assert (stack_pointer + articles.length) < stack.length;
        System.arraycopy(articles, 0, stack, stack_pointer+1,
                           articles.length);
        stack_pointer += articles.length;
    }
}
```

- The new version of Stack works fine; in fact, it's better than the previous version. Unfortunately, the Monitorable_stack derived class doesn't work any more, since it won't correctly track stack usage if push_many() is called (the derived-class version of push() is no longer called by the inherited push_many() method, so push_many() no longer updates the high_water_mark). Stack is a fragile base class.

Summary

- In general, it's best to avoid concrete base classes and extends relationships in favor of interfaces and implements relationships.

Listing 0.1. Eliminate fragile base classes using interfaces

```
1| import java.util.*;
2|
3| interface Stack
4| {
5|     void push( Object o );
6|     Object pop();
7|     void push_many( Object[] source );
8| }
9|
10| class Simple_stack implements Stack
11| {   private int stack_pointer = -1;
12|     private Object[] stack = new Object[1000];
13|
14|     public void push( Object o )
15|     {   assert stack_pointer < stack.length;
16|
17|         stack[ ++stack_pointer ] = o;
18|     }
19|
20|     public Object pop()
21|     {   assert stack_pointer >= 0;
22|
23|         return stack[ stack_pointer-- ];
24|     }
25|
26|     public void push_many( Object[] source )
27|     {   assert (stack_pointer + source.length) < stack.length;
28|
29|         System.arraycopy(source, 0, stack, stack_pointer+1, source.length);
30|         stack_pointer += source.length;
31|     }
32| }
```

```
33|
34|
35| class Monitorable_Stack implements Stack
36| {
37|     private int high_water_mark = 0;
38|     private int current_size;
39|     Simple_stack stack = new Simple_stack();
40|
41|     public void push( Object o )
42|     {   if( ++current_size > high_water_mark )
43|         high_water_mark = current_size;
44|         stack.push(o);
45|     }
46|
47|     public Object pop()
48|     {   --current_size;
49|         return stack.pop();
50|     }
51|
52|     public void push_many( Object[] source )
53|     {
54|         if( current_size + source.length > high_water_mark )
55|             high_water_mark = current_size + source.length;
56|
57|         stack.push_many( source );
58|     }
59|
60|     public int maximum_size()
61|     {   return high_water_mark;
62|     }
63| }
64|
```

Invokevirtual vs invokeinterface

Java

- 编译期
 - 静态
 - 多分派
 - overloading
- 运行期
 - 动态
 - 单分派
 - overriding
- 都是invokevirtual指令

```
public class Dispatch {

    static class QQ{}
    static class _360{}
    public static class Father{

        public void hardChoice(_360 _360) {
            System.out.println("Father choose 360");
        }

        public void hardChoice(QQ qq) {
            System.out.println("Father choose qq");
        }

    }

    public static class Son extends Father{

        public void hardChoice(_360 _360) {
            System.out.println("Son choose 360");
        }

        public void hardChoice(QQ qq) {
            System.out.println("Son choose qq");
        }

    }

    public static void main(String[] args) {
        Father father = new Father();
        Father son = new Son();
        father.hardChoice(new _360());
        son.hardChoice(new QQ());
    }

}
```

```

Classfile /Users/qinliu/Dispatch.class
Last modified 2015-5-20; size 569 bytes
MD5 checksum 04224e8234feafca65ece76dcc3fd6c6
Compiled from "Dispatch.java"
public class Dispatch
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Methodref          #13.#27      // java/lang/Object."<init>":()V
  #2 = Class               #28          // Dispatch$Father
  #3 = Methodref          #2.#27        // Dispatch$Father."<init>":()V
  #4 = Class               #29          // Dispatch$Son
  #5 = Methodref          #4.#27        // Dispatch$Son."<init>":()V
  #6 = Class               #30          // Dispatch$_360
  #7 = Methodref          #6.#27        // Dispatch$_360."<init>":()V
  #8 = Methodref          #2.#31        // Dispatch$Father.hardChoice:(LDispatch$_360;)V
  #9 = Class               #32          // Dispatch$QQ
 #10 = Methodref          #9.#27        // Dispatch$QQ."<init>":()V
 #11 = Methodref          #2.#33        // Dispatch$Father.hardChoice:(LDispatch$QQ;)V
 #12 = Class               #34          // Dispatch
 #13 = Class               #35          // java/lang/Object
 #14 = Utf8                Son
 #15 = Utf8                InnerClasses
 #16 = Utf8                Father
 #17 = Utf8                _360
 #18 = Utf8                QQ
 #19 = Utf8                <init>
 #20 = Utf8                ()V
 #21 = Utf8                Code
 #22 = Utf8                LineNumberTable
 #23 = Utf8                main
 #24 = Utf8                ([Ljava/lang/String;)V
 #25 = Utf8                SourceFile
 #26 = Utf8                Dispatch.java
 #27 = NameAndType          #19:#20      // "<init>":()V
 #28 = Utf8                Dispatch$Father
 #29 = Utf8                Dispatch$Son
 #30 = Utf8                Dispatch$_360
 #31 = NameAndType          #36:#37      // hardChoice:(LDispatch$_360;)V
 #32 = Utf8                Dispatch$QQ
 #33 = NameAndType          #36:#38      // hardChoice:(LDispatch$QQ;)V
 #34 = Utf8                Dispatch
 #35 = Utf8                java/lang/Object
 #36 = Utf8                hardChoice
 #37 = Utf8                (LDispatch$_360;)V
 #38 = Utf8                (LDispatch$QQ;)V

```



```

public Dispatch();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=1, locals=1, args_size=1
      0: aload_0
      1: invokespecial #1                  // Method java/lang/Object."<init>":()V
      4: return
  LineNumberTable:
    line 1: 0
    line 17: 4

public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=3, args_size=1
      0: new          #2                  // class Dispatch$Father
      3: dup
      4: invokespecial #3                  // Method Dispatch$Father."<init>":()V
      7: astore_1
      8: new          #4                  // class Dispatch$Son
     11: dup
     12: invokespecial #5                  // Method Dispatch$Son."<init>":()V
     15: astore_2
     16: aload_1
     17: new          #6                  // class Dispatch$_360
     20: dup
     21: invokespecial #7                  // Method Dispatch$_360."<init>":()V
     24: invokevirtual #8                  // Method Dispatch$Father.hardChoice:(LDispatch$_360;)V
     27: aload_2
     28: new          #9                  // class Dispatch$QQ
     31: dup
     32: invokespecial #10                 // Method Dispatch$QQ."<init>":()V
     35: invokevirtual #11                 // Method Dispatch$Father.hardChoice:(LDispatch$QQ;)V
     38: return
  LineNumberTable:
    line 30: 0
    line 31: 8
    line 32: 16
    line 33: 27
    line 34: 38
}
SourceFile: "Dispatch.java"
InnerClasses:
  public static #14= #4 of #12; //Son=class Dispatch$Son of class Dispatch
  public static #16= #2 of #12; //Father=class Dispatch$Father of class Dispatch
  static #17= #6 of #12; //_360=class Dispatch$_360 of class Dispatch
  static #18= #9 of #12; //QQ=class Dispatch$QQ of class Dispatch

```

分析

- 输出结果：

- Father choose 360

- Son choose qq

- 分析

- ```
Father father = new Father();
```

- ```
Father son = new Son();
```

-

- ```
/**
```

- ```
    * Father里面有两个重载的hardChoice方法。所以会根据hardChoice()的实参的【静态类型】来决定调用哪个版本的方法。
```

- ```
 */
```

- ```
    father.hardChoice(new _360());
```

-

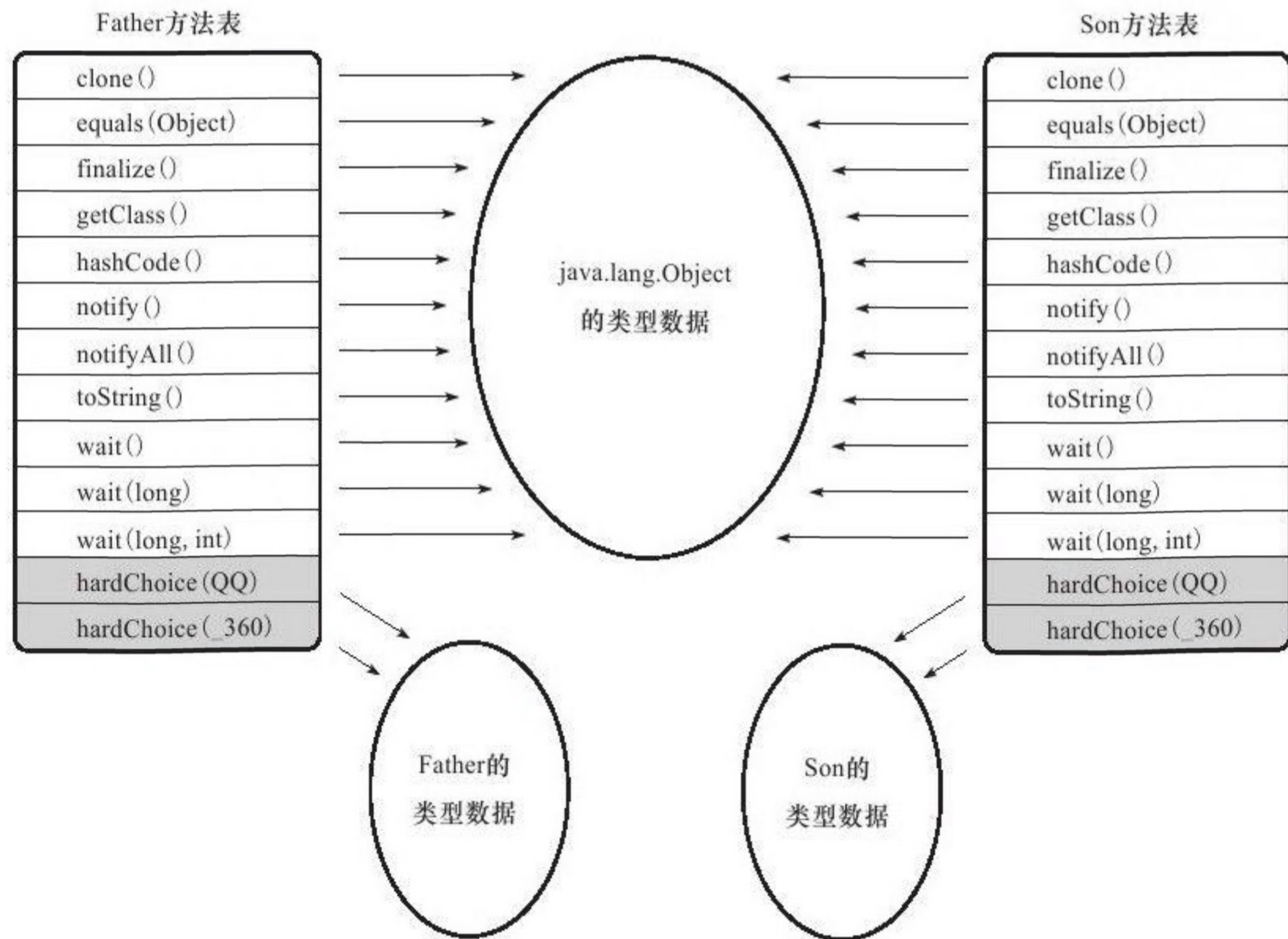
- ```
/**
```

- ```
    * 变量son的静态类型是Father，实际类型是Son。并且类Son重写了父类Father里面的两个重载的hardChoice方法。
```

- ```
 * 所以运行的时候首先会确定调用子类Son里面的方法，然后在根据hardChoice()的实参的【静态类型】来决定调用Son里面的哪个版本的方法。
```

- ```
    */
```

- ```
 son.hardChoice(new QQ());
```



# 实现动态分派时的方法表结构

Each Java class is associated with a virtual method table that contains "links" to the bytecode of each method of a class. That table is inherited from the superclass of a particular class and extended with regard to the new methods of a subclass. E.g.,

```
class BaseClass {
 public void method1() { }
 public void method2() { }
 public void method3() { }
}

class NextClass extends BaseClass {
 public void method2() { } // overridden from BaseClass
 public void method4() { }
}
```

results in the tables

BaseClass

1. BaseClass/method1()
2. BaseClass/method2()
3. BaseClass/method3()

NextClass

1. BaseClass/method1()
2. NextClass/method2()
3. BaseClass/method3()
4. NextClass/method4()

Note, how the virtual method table of NextClass retains the order of entries of the table of BaseClass and just overwrites the "link" of method2() which it overrides.

An implementation of the JVM can thus optimize a call to `invokevirtual` by remembering that `BaseClass/method3()` will always be the third entry in the virtual method table of any object this method will ever be invoked on.



With `invokeinterface` this optimization is not possible. E.g.,

```
interface MyInterface {
 void ifaceMethod();
}

class AnotherClass extends NextClass implements MyInterface {
 public void method4() { } // overridden from NextClass
 public void ifaceMethod() { }
}
```

```
class MyClass implements MyInterface {
 public void method5() { }
 public void ifaceMethod() { }
}
```

This class hierarchy results in the virtual method tables

- AnotherClass
- 1. BaseClass/method1()
  - 2. NextClass/method2()
  - 3. BaseClass/method3()
  - 4. AnotherClass/method4()
  - 5. MyInterface/ifaceMethod()

- MyClass
- 1. MyClass/method5()
  - 2. MyInterface/ifaceMethod()

As you can see, AnotherClass contains the interface's method in its fifth entry and MyClass contains it in its second entry. To actually find the correct entry in the virtual method table, a call to a method with `invokeinterface` will always have to search the complete table without a chance for the style of optimization that `invokevirtual` does.

```

public class InvokevirtualVsInvokeinterface {
 private static interface I {
 public int getInteger ();
 }

 private static class A implements I {
 public int getInteger () { return 0; }
 }

 private static class B extends A { }

 static volatile I i = new B();
 static volatile A a = new B();

 public static void main(String[] args) {
 {
 long tm1 = System.nanoTime();
 for (int k = 0; k < 1000000000; ++k) {
 a.getInteger();
 }
 long tm2 = System.nanoTime();
 System.out.println("invokevirtual took " + (Math.abs(tm2 - tm1) / 1000) + " us");
 }

 {
 long tm1 = System.nanoTime();
 for (int k = 0; k < 1000000000; ++k) {
 i.getInteger();
 }
 long tm2 = System.nanoTime();
 System.out.println("invokeinterface took " + (Math.abs(tm2 - tm1) / 1000) + " us");
 }

 // Output on Intel Xeon X5570 @ 2.93GHz:
 // invokevirtual took 41170 us
 // invokeinterface took 66305 us
 }
}

```

```
Compiled from "InvokevirtualVsInvokeinterface.java"
public class InvokevirtualVsInvokeinterface extends java.lang.Object{
static volatile InvokevirtualVsInvokeinterface$I i;
```

```
static volatile InvokevirtualVsInvokeinterface$A a;
```

```
public InvokevirtualVsInvokeinterface();
Code:
0: aload_0
1: invokespecial#1; //Method java/lang/Object."<init>":()V
4: return
```

```
public static void main(java.lang.String[]);
Code:
0: invokestatic #2; //Method java/lang/System.nanoTime:()J
3: lstore_1
4: iconst_0
5: istore_3
6: iload_3
7: ldc #3; //int 1000000000
9: if_icmpge 25
12: getstatic #4; //Field a:LInvokevirtualVsInvokeinterface$A;
15: invokevirtual #5; //Method InvokevirtualVsInvokeinterface$A.getInteger:()I
18: pop
19: iinc 3, 1
22: goto 6
25: invokestatic #2; //Method java/lang/System.nanoTime:()J
28: lstore_3
29: getstatic #6; //Field java/lang/System.out:Ljava/io/PrintStream;
32: new #7; //class java/lang/StringBuilder
35: dup
36: invokespecial#8; //Method java/lang/StringBuilder."<init>":()V
39: ldc #9; //String invokevirtual took
41: invokevirtual #10; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
44: lload_3
45: lload_1
46: lsub
47: invokestatic #11; //Method java/lang/Math.abs:(J)J
50: ldc2_w #12; //long 1000l
53: ldiv
54: invokevirtual #14; //Method java/lang/StringBuilder.append:(J)Ljava/lang/StringBuilder;
57: ldc #15; //String us
59: invokevirtual #10; //Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
62: invokevirtual #16; //Method java/lang/StringBuilder.toString:()Ljava/lang/String;
65: invokevirtual #17; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
68: invokestatic #2; //Method java/lang/System.nanoTime:()J
```

```
71: lstore_1
72: iconst_0
73: istore_3
74: iload_3
75: ldc #3; //int 1000000000
77: if_icmpge 95
80: getstatic #18; //Field i:LInvokevirtualVsInvokeinterface$I;
83: invokeinterface #19, 1; //InterfaceMethod
InvokevirtualVsInvokeinterface$I.getInteger():I
88: pop
89: iinc 3, 1
92: goto 74
95: invokestatic #2; //Method java/lang/System.nanoTime():J
98: lstore_3
99: getstatic #6; //Field java/lang/System.out:Ljava/io/PrintStream;
102: new #7; //class java/lang/StringBuilder
105: dup
106: invokespecial #8; //Method
java/lang/StringBuilder."<init>":()V
109: ldc #20; //String invokeinterface took
111: invokevirtual #10; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilde
r;
114: lload_3
115: lload_1
116: lsub
117: invokestatic #11; //Method java/lang/Math.abs:(J)J
120: ldc2_w #12; //long 1000l
123: ldiv
124: invokevirtual #14; //Method
java/lang/StringBuilder.append:(J)Ljava/lang/StringBuilder;
127: ldc #15; //String us
```

```
129: invokevirtual #10; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilde
r;
132: invokevirtual #16; //Method
java/lang/StringBuilder.toString():Ljava/lang/String;
135: invokevirtual #17; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
138: return

static {};
Code:
0: new #21; //class InvokevirtualVsInvokeinterface$B
3: dup
4: aconst_null
5: invokespecial #22; //Method
InvokevirtualVsInvokeinterface$B."<init>":(LInvokevirtualVsInvokeinterface
$1;)V
8: putstatic #18; //Field i:LInvokevirtualVsInvokeinterface$I;
11: new #21; //class InvokevirtualVsInvokeinterface$B
14: dup
15: aconst_null
16: invokespecial #22; //Method
InvokevirtualVsInvokeinterface$B."<init>":(LInvokevirtualVsInvokeinterface
$1;)V
19: putstatic #4; //Field a:LInvokevirtualVsInvokeinterface$A;
22: return

}
```



- A benchmark code to compare invokevirtual with invokeinterface performance. **invokeinterface is 38% slower.**

```
class Base {

public:

virtual void f() { cout << "Base::f" << endl; }

virtual void g() { cout << "Base::g" << endl; }

virtual void h() { cout << "Base::h" << endl; }

};
```

按照上面的说法，我们可以通过Base的实例来得到虚函数表。下面是实际例程：

```
typedef void(*Fun)(void);

Base b;

Fun pFun = NULL;

cout << "虚函数表地址：" << (int*)&b << endl;

cout << "虚函数表 — 第一个函数地址：" << (int*)(int*)&b << endl;

// Invoke the first virtual function

pFun = (Fun*)((int*)(int*)&b);

pFun();
```

实际运行经果如下：(Windows XP+VS2003, Linux 2.6.22 + GCC 4.1.3)

虚函数表地址：0012FED4

虚函数表 — 第一个函数地址：0044F148

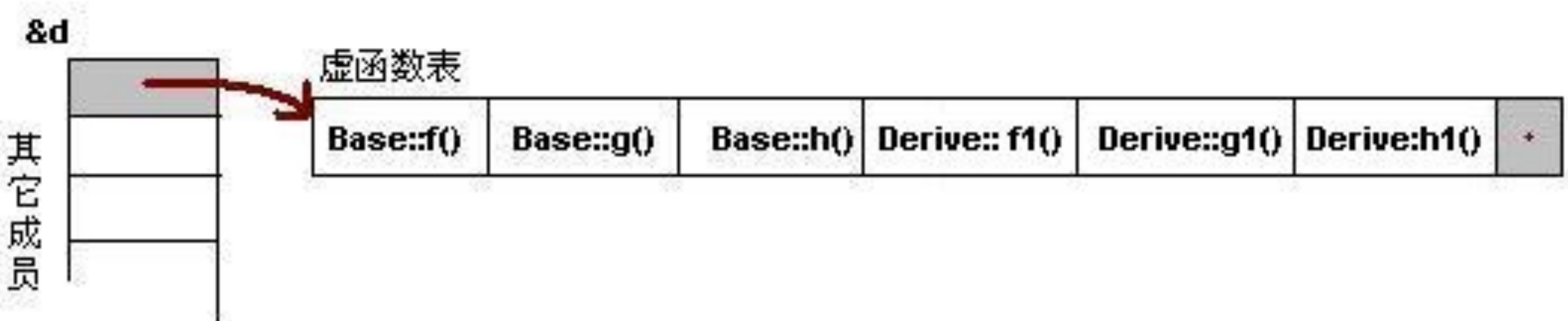
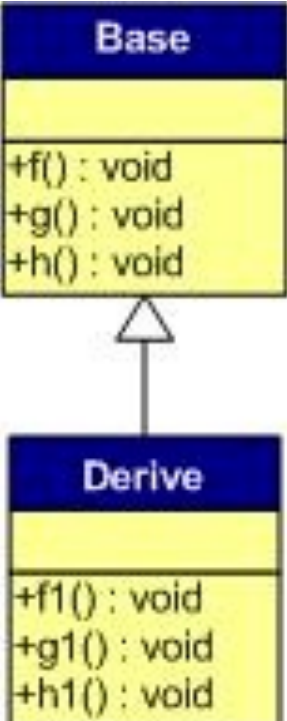
Base::f

# C++虚函数的实现

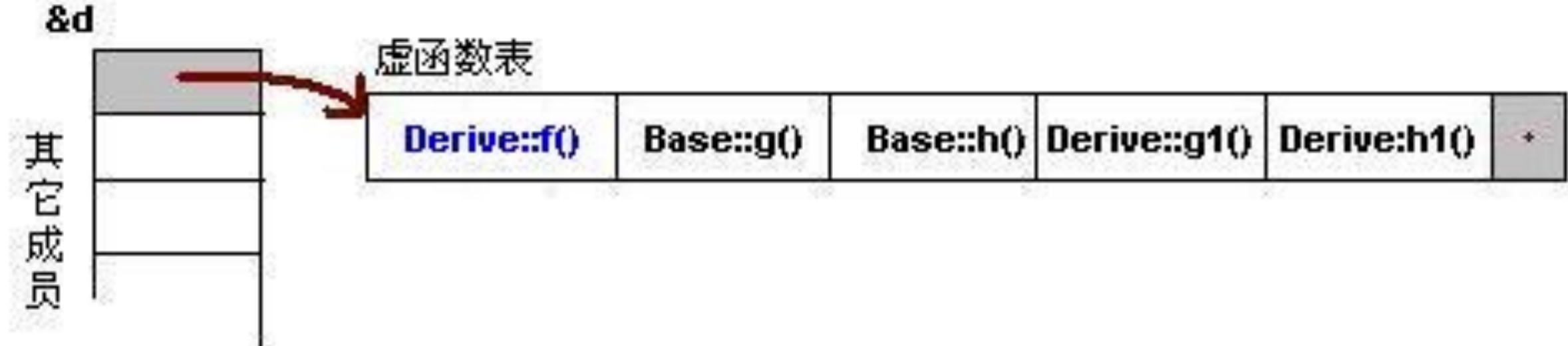
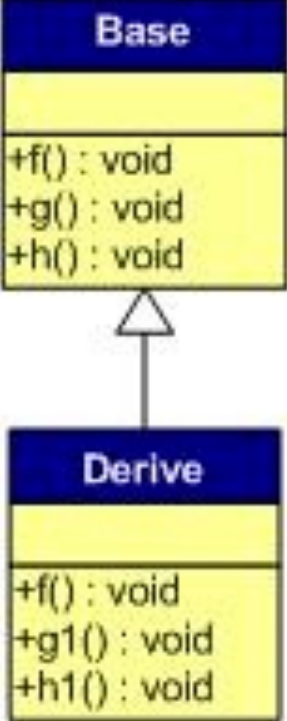
基类



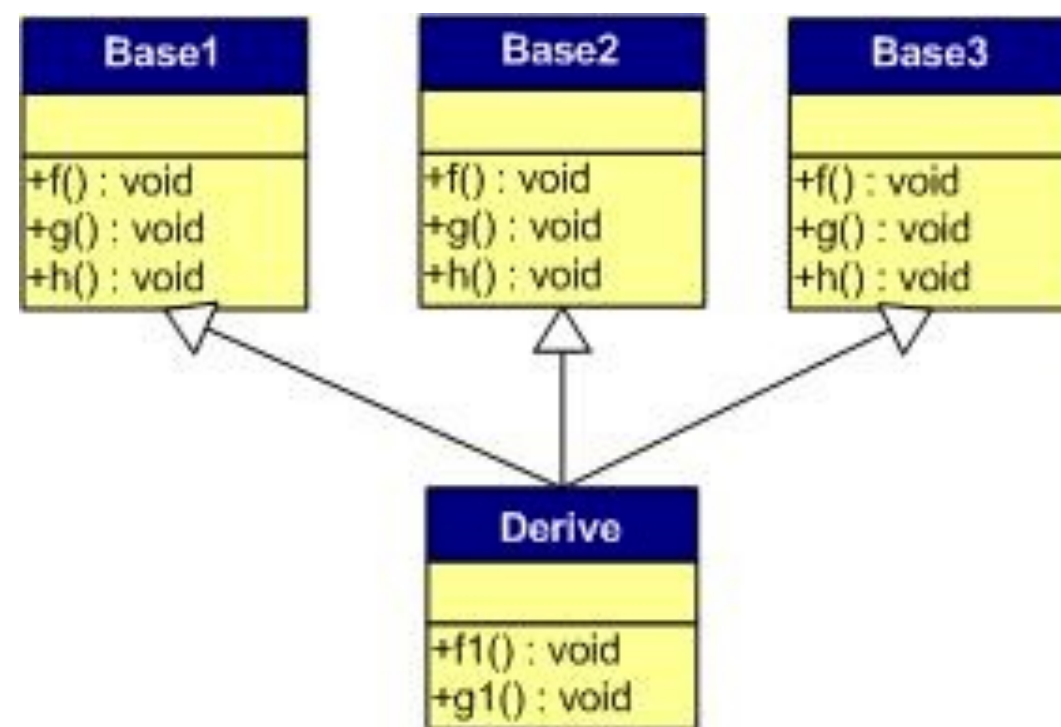
无虚函数覆盖



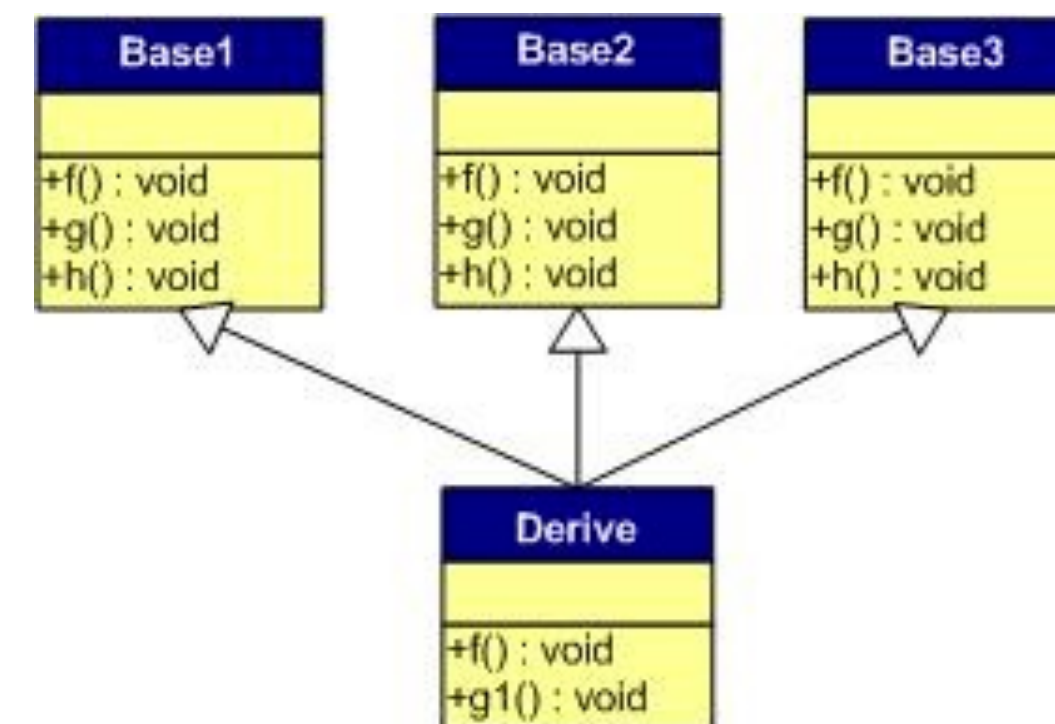
虚函数覆盖



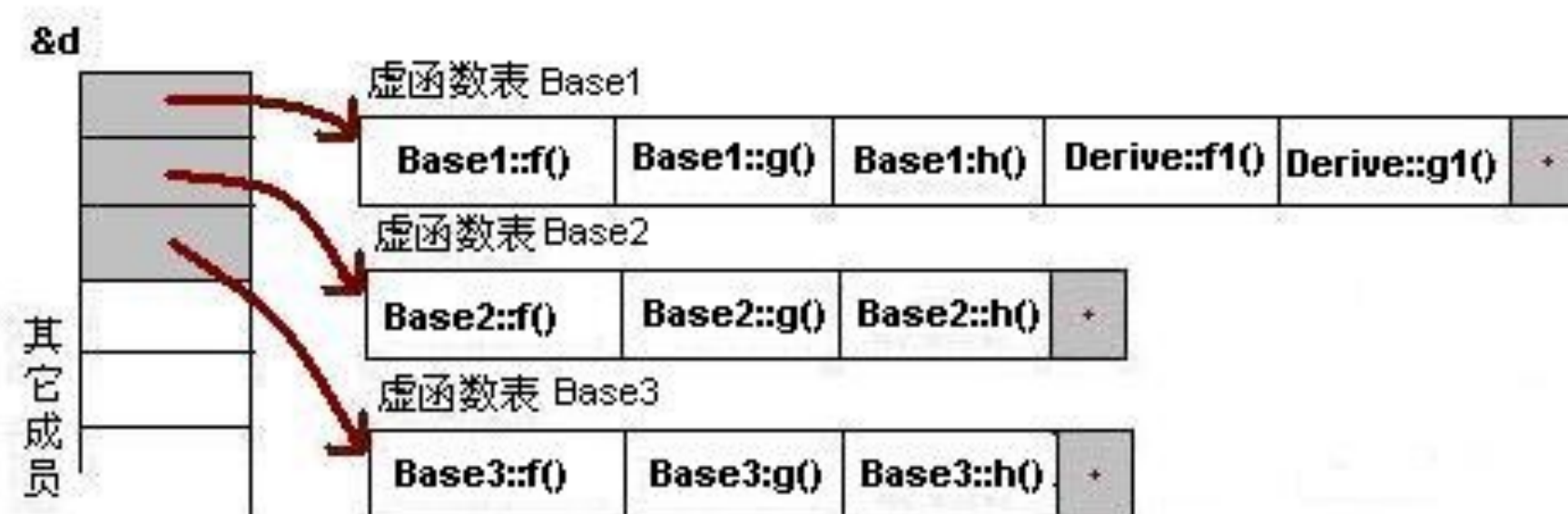
虚函数覆盖



无虚函数覆盖



虚函数覆盖



# 多重继承

# default method in Java 8



# 一个简单的例子

- interface InterfaceA {
- default void foo() {
- System.out.println("InterfaceA foo");
- }
- }
- 
- class ClassA implements InterfaceA {
- }
- 
- public class Test {
- public static void main(String[] args) {
- new ClassA().foo(); // 打印 : “InterfaceA foo”
- }
- }

```
promote:Downloads liuqin$ javap -verbose Test
Classfile /Users/liuqin/Downloads/Test.class
 Last modified 2016-5-19; size 300 bytes
 MD5 checksum c5c350a838c93cf66c59d12a8a4a32b0
 Compiled from "Test.java"
public class Test
 minor version: 0
 major version: 52
 flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
 #1 = Methodref #6.#15 //
java/lang/Object."<init>":()V
 #2 = Class #16 // ClassA
 #3 = Methodref #2.#15 //
ClassA."<init>":()V
 #4 = Methodref #2.#17 // ClassA.foo:()V
 #5 = Class #18 // Test
 #6 = Class #19 //
java/lang/Object
 #7 = Utf8 <init>
 #8 = Utf8 ()V
 #9 = Utf8 Code
#10 = Utf8 LineNumberTable
#11 = Utf8 main
#12 = Utf8 ([Ljava/lang/String;)V
#13 = Utf8 SourceFile
#14 = Utf8 Test.java
#15 = NameAndType #7:#8 // "<init>":()V
#16 = Utf8 ClassA
#17 = NameAndType #20:#8 // foo:()V
#18 = Utf8 Test
#19 = Utf8 java/lang/Object
#20 = Utf8 foo
{
 public Test();
 descriptor: ()V
```

```
flags: ACC_PUBLIC
Code:
 stack=1, locals=1, args_size=1
 0: aload_0
 1: invokespecial #1 // Method
java/lang/Object."<init>":()V
 4: return
LineNumberTable:
 line 10: 0

 public static void main(java.lang.String[]);
 descriptor: ([Ljava/lang/String;)V
 flags: ACC_PUBLIC, ACC_STATIC
 Code:
 stack=2, locals=1, args_size=1
 0: new #2 // class
 3: dup
 4: invokespecial #3 // Method
ClassA."<init>":()V
 7: invokevirtual #4 // Method
ClassA.foo:()V
 10: return
 LineNumberTable:
 line 12: 0
 line 13: 10
 }
 SourceFile: "Test.java"
```

# 为什么要有默认方法

- 在 java 8 之前，接口与其实现类之间的 耦合度 太高了（tightly coupled），当需要为一个接口添加方法时，所有的实现类都必须随之修改。默认方法解决了这个问题，它可以为接口添加新的方法，而不会破坏已有的接口的实现。这在 lambda 表达式作为 java 8 语言的重要特性而出现之际，为升级旧接口且保持向后兼容（backward compatibility）提供了途径。
- ```
String[] array = new String[] {  
    "hello",  
    ", ",  
    "world",  
};  
  
List<String> list = Arrays.asList(array);  
  
list.forEach(System.out::println); // 这是 jdk 1.8 新增的接口默认方法
```

jdk 1.8 的 Iterable 接口中的 forEach 默认方法：

- package java.lang;
- import java.util.Objects;
- import java.util.function.Consumer;
- public interface Iterable<T> {
- default void forEach(Consumer<? super T> action) {
- Objects.requireNonNull(action);
- for (T t : this) {
- action.accept(t);
- }
- }
- }

默认方法的继承

```
interface InterfaceA {  
    default void foo() {  
        System.out.println("InterfaceA foo");  
    }  
}
```

```
interface InterfaceB extends InterfaceA {  
}
```

```
interface InterfaceC extends InterfaceA {  
    @Override  
    default void foo() {  
        System.out.println("InterfaceC foo");  
    }  
}
```

```
interface InterfaceD extends InterfaceA {  
    @Override  
    void foo();  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        new InterfaceB().foo(); // 打印 : "InterfaceA foo"  
        new InterfaceC().foo(); // 打印 : "InterfaceC foo"  
        new InterfaceD() {  
            @Override  
            public void foo() {  
                System.out.println("InterfaceD foo");  
            }  
        }.foo(); // 打印 : "InterfaceD foo"  
  
        // 或者使用 lambda 表达式  
        ((InterfaceD) () -> System.out.println("InterfaceD foo")).foo()  
    }  
}
```


默认方法的多重继承

```
interface InterfaceA {  
    default void foo() {  
        System.out.println("InterfaceA foo");  
    }  
}
```

```
interface InterfaceB {  
    default void bar() {  
        System.out.println("InterfaceB bar");  
    }  
}
```

```
interface InterfaceC {  
    default void foo() {  
        System.out.println("InterfaceC foo");  
    }  
  
    default void bar() {  
        System.out.println("InterfaceC bar");  
    }  
}
```

```
class ClassA implements InterfaceA, InterfaceB {  
}
```

// 错误

```
//class ClassB implements InterfaceB, InterfaceC {  
//}
```

```
class ClassB implements InterfaceB, InterfaceC {  
    @Override  
    public void bar() {  
        InterfaceB.super.bar(); // 调用 InterfaceB 的 bar 方法  
        InterfaceC.super.bar(); // 调用 InterfaceC 的 bar 方法  
        System.out.println("ClassB bar"); // 做其他的事  
    }  
}
```

```

promote:Downloads liuqin$ javap -verbose ClassB
Classfile /Users/liuqin/Downloads/ClassB.class
  Last modified 2016-5-19; size 457 bytes
  MD5 checksum 3ce098725d3829e25f1f22f48fbb014a
  Compiled from "ClassB.java"
class ClassB implements InterfaceB,InterfaceC
  minor version: 0
  major version: 52
  flags: ACC_SUPER
Constant pool:
  #1 = Methodref      #8.#18      // java/lang/Object."<init>":()V
  #2 = InterfaceMethodref #9.#19    // InterfaceB.bar:()V
  #3 = InterfaceMethodref #10.#19   // InterfaceC.bar:()V
  #4 = Fieldref       #20.#21      //
java/lang/System.out:Ljava/io/PrintStream;
  #5 = String         #22          // ClassB bar
  #6 = Methodref      #23.#24      //
java/io/PrintStream.println:(Ljava/lang/String;)V
  #7 = Class          #25          // ClassB
  #8 = Class          #26          // java/lang/Object
  #9 = Class          #27          // InterfaceB
  #10 = Class         #28          // InterfaceC
  #11 = Utf8          <init>
  #12 = Utf8          ()V
  #13 = Utf8          Code
  #14 = Utf8          LineNumberTable
  #15 = Utf8          bar
  #16 = Utf8          SourceFile
  #17 = Utf8          ClassB.java
  #18 = NameAndType   #11:#12      // "<init>":()V
  #19 = NameAndType   #15:#12      // bar:()V
  #20 = Class         #29          // java/lang/System
  #21 = NameAndType   #30:#31      // out:Ljava/io/PrintStream;
  #22 = Utf8          ClassB bar
  #23 = Class         #32          // java/io/PrintStream
  #24 = NameAndType   #33:#34      // println:(Ljava/lang/String;)V
  #25 = Utf8          ClassB
  #26 = Utf8          java/lang/Object
  #27 = Utf8          InterfaceB
  #28 = Utf8          InterfaceC
  #29 = Utf8          java/lang/System
  #30 = Utf8          out
  #31 = Utf8          Ljava/io/PrintStream;
  #32 = Utf8          java/io/PrintStream
  #33 = Utf8          println
  #34 = Utf8          (Ljava/lang/String;)V
{
  ClassB();

```

```

descriptor: ()V
flags:
Code:
  stack=1, locals=1, args_size=1
    0: aload_0
    1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
    4: return
  LineNumberTable:
    line 30: 0

  public void bar();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=1, args_size=1
      0: aload_0
      1: invokespecial #2                  // InterfaceMethod
InterfaceB.bar:()V
      4: aload_0
      5: invokespecial #3                  // InterfaceMethod
InterfaceC.bar:()V
      8: getstatic      #4                  // Field
java/lang/System.out:Ljava/io/PrintStream;
     11: ldc            #5                  // String ClassB bar
     13: invokevirtual #6                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
     16: return
  LineNumberTable:
    line 33: 0
    line 34: 4
    line 35: 8
    line 36: 16
}
SourceFile: "ClassB.java"

```

接口继承行为发生冲突时的解决规则

```
interface InterfaceA {  
    default void foo() {  
        System.out.println("InterfaceA foo");  
    }  
}  
  
interface InterfaceB extends InterfaceA {  
    @Override  
    default void foo() {  
        System.out.println("InterfaceB foo");  
    }  
}  
  
// 正确  
class ClassA implements InterfaceA, InterfaceB {  
}  
  
class ClassB implements InterfaceA, InterfaceB {  
    @Override  
    public void foo() {  
        // InterfaceA.super.foo(); // 错误  
        InterfaceB.super.foo();  
    }  
}
```

接口与抽象类

```
interface InterfaceA {  
    default void foo() {  
        System.out.println("InterfaceA foo");  
    }  
  
    default void bar() {  
        System.out.println("InterfaceA bar");  
    }  
}
```

```
abstract class AbstractClassA {  
    public abstract void foo();  
  
    public void bar() {  
        System.out.println("AbstractClassA bar");  
    }  
}
```

```
class ClassA extends AbstractClassA implements InterfaceA {  
    @Override  
    public void foo() {  
        InterfaceA.super.foo();  
    }  
}
```

当接口继承行为发生冲突时的另一个规则是，类的方法声明优先于接口默认方法，无论该方法具体的还是抽象的。

```
public class Test {  
    public static void main(String[] args) {  
        ClassA classA = new ClassA();  
        classA.foo(); // 打印：“InterfaceA foo”  
        classA.bar(); // 打印：“AbstractClassA bar”  
    }  
}
```

```
promote:Downloads liuqin$ javap -verbose Test2
Classfile /Users/liuqin/Downloads/Test2.class
  Last modified 2016-5-19; size 332 bytes
  MD5 checksum 502a6670288dd911087660d26e2ff3ad
  Compiled from "Test2.java"
public class Test2
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Methodref      #7.#16      // java/lang/Object."<init>":()V
  #2 = Class          #17          // ClassA
  #3 = Methodref      #2.#16      // ClassA."<init>":()V
  #4 = Methodref      #2.#18      // ClassA.foo:()V
  #5 = Methodref      #2.#19      // ClassA.bar:()V
  #6 = Class          #20          // Test2
  #7 = Class          #21          // java/lang/Object
  #8 = Utf8           <init>
  #9 = Utf8           ()V
 #10 = Utf8           Code
 #11 = Utf8           LineNumberTable
 #12 = Utf8           main
 #13 = Utf8           ([Ljava/lang/String;)V
 #14 = Utf8           SourceFile
 #15 = Utf8           Test2.java
 #16 = NameAndType    #8:#9        // "<init>":()V
 #17 = Utf8           ClassA
 #18 = NameAndType    #22:#9       // foo:()V
 #19 = NameAndType    #23:#9       // bar:()V
 #20 = Utf8           Test2
 #21 = Utf8           java/lang/Object
 #22 = Utf8           foo
 #23 = Utf8           bar
{
  public Test2();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
        0: aload_0
        1: invokespecial #1              // Method
java/lang/Object."<init>":()V
        4: return
    LineNumberTable:
      line 26: 0
}
```

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=1
      0: new                #2              // class ClassA
      3: dup
      4: invokespecial #3              // Method ClassA."<init>":()V
      7: astore_1
      8: aload_1
      9: invokevirtual #4              // Method ClassA.foo:()V
     12: aload_1
     13: invokevirtual #5              // Method ClassA.bar:()V
     16: return
  LineNumberTable:
    line 28: 0
    line 29: 8
    line 30: 12
    line 31: 16
}
SourceFile: "Test2.java"
```


接口和抽象方法不可相互替代

- 虽然 Java 8 的接口的默认方法就像抽象类，能提供方法的实现，但是他们俩仍然是不可相互代替的：
- 接口可以被类多实现（被其他接口多继承），抽象类只能被单继承。
- 接口中没有 this 指针，没有构造函数，不能拥有实例字段（实例变量）或实例方法，无法保存状态（state），抽象方法中可以。
- 抽象类不能在 java 8 的 lambda 表达式中使用。
- 从设计理念上，接口反映的是“like-a”关系，抽象类反映的是“is-a”关系

接口的静态方法

- interface InterfaceA {
- default void foo() {
- printHelloWorld();
- }
-
- static void printHelloWorld() {
- System.out.println("hello, world");
- }
- }
-
- public class Test {
- public static void main(String[] args) {
- InterfaceA.printHelloWorld(); // 打印 : “hello, world”
- }
- }

其他注意点

- default 关键字只能在接口中使用（以及用在 switch 语句的 default 分支），不能用在抽象类中。
- 接口默认方法不能覆写 Object 类的 equals、hashCode 和 toString 方法。
- 接口中的静态方法必须是 public 的，public 修饰符可以省略，static 修饰符不能省略。
- 即使使用了 java 8 的环境，一些 IDE 仍然可能在一些代码的实时编译提示时出现异常的提示（例如无法发现 java 8 的语法错误），因此不要过度依赖 IDE。