IT服务中国

软件架构

Software Architecture

叶红星　　博士

技术总监

yehx@DigitalChina.com

神州数码系统集成服务有限公司培训中心

神州数码 Digital China

IT服务中国

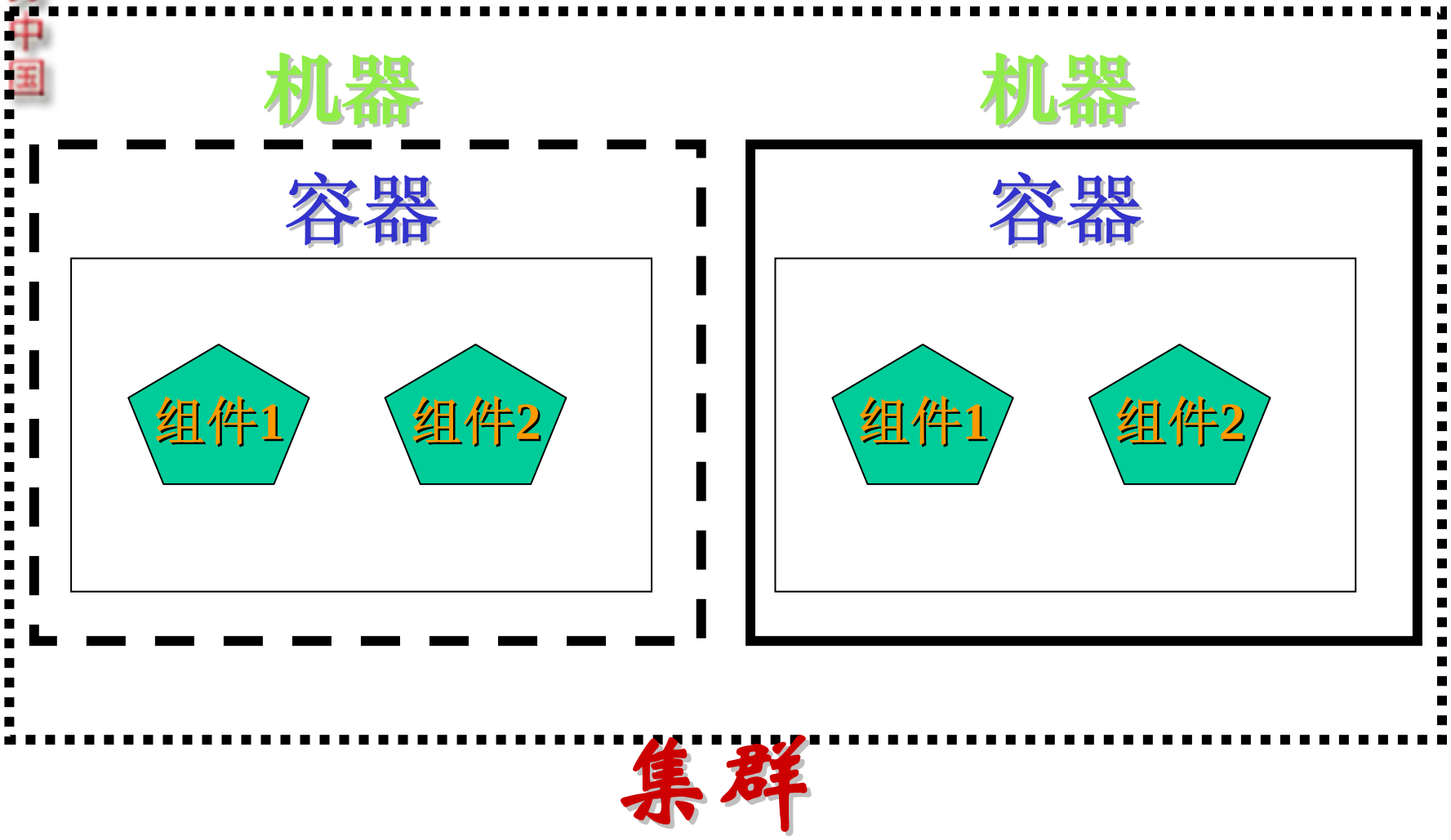
架构实施

神州数码
Digital China

目标

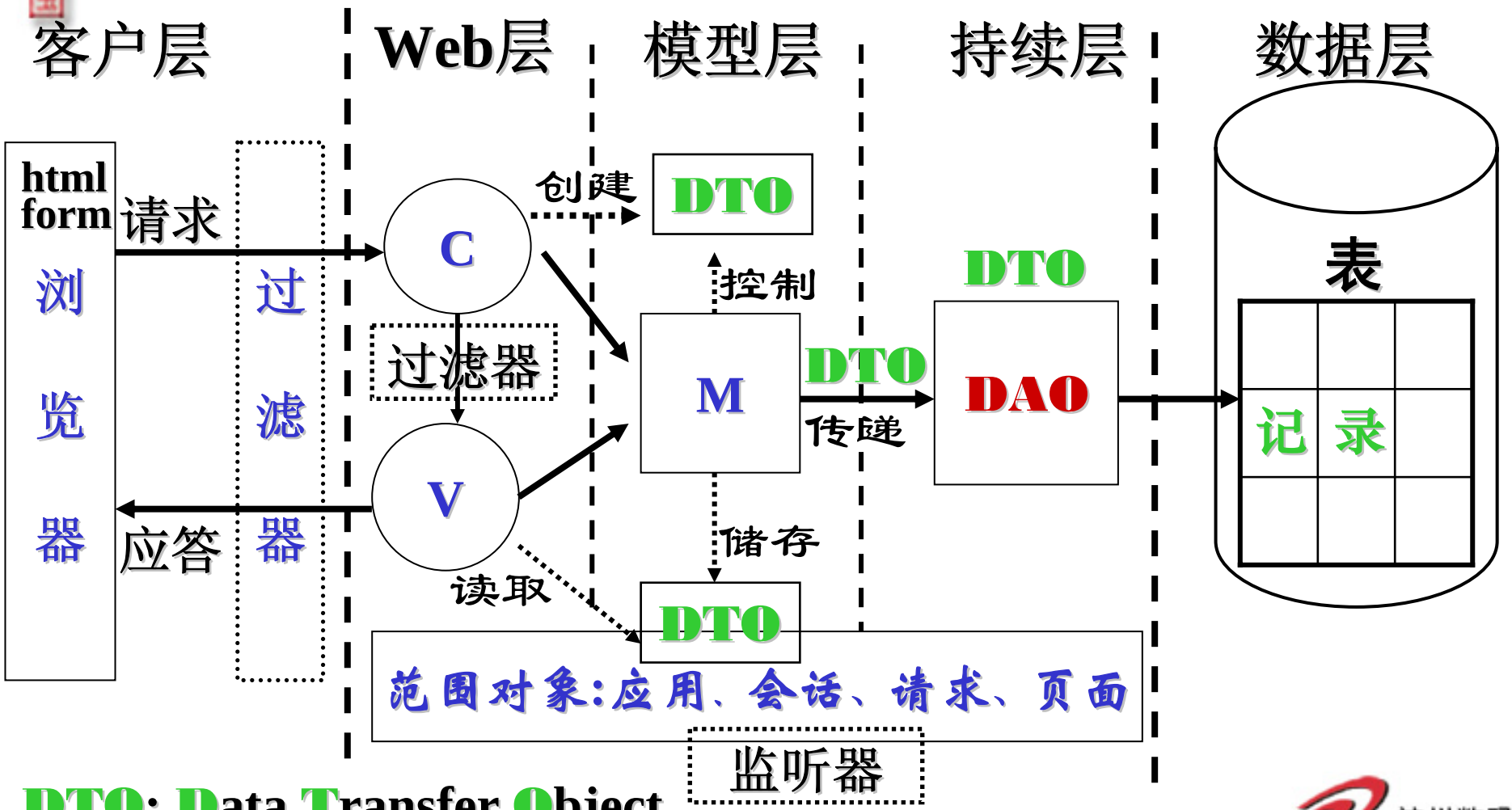
- 深刻理解多态
- 熟练掌握分层和MVC架构风格
- 深刻理解Java EE中的一些架构模式
- 深刻理解Java的内存模型

Java EE的CBD核心概念模型



RequestDispatcher

B/S应用的分层和MVC架构模式

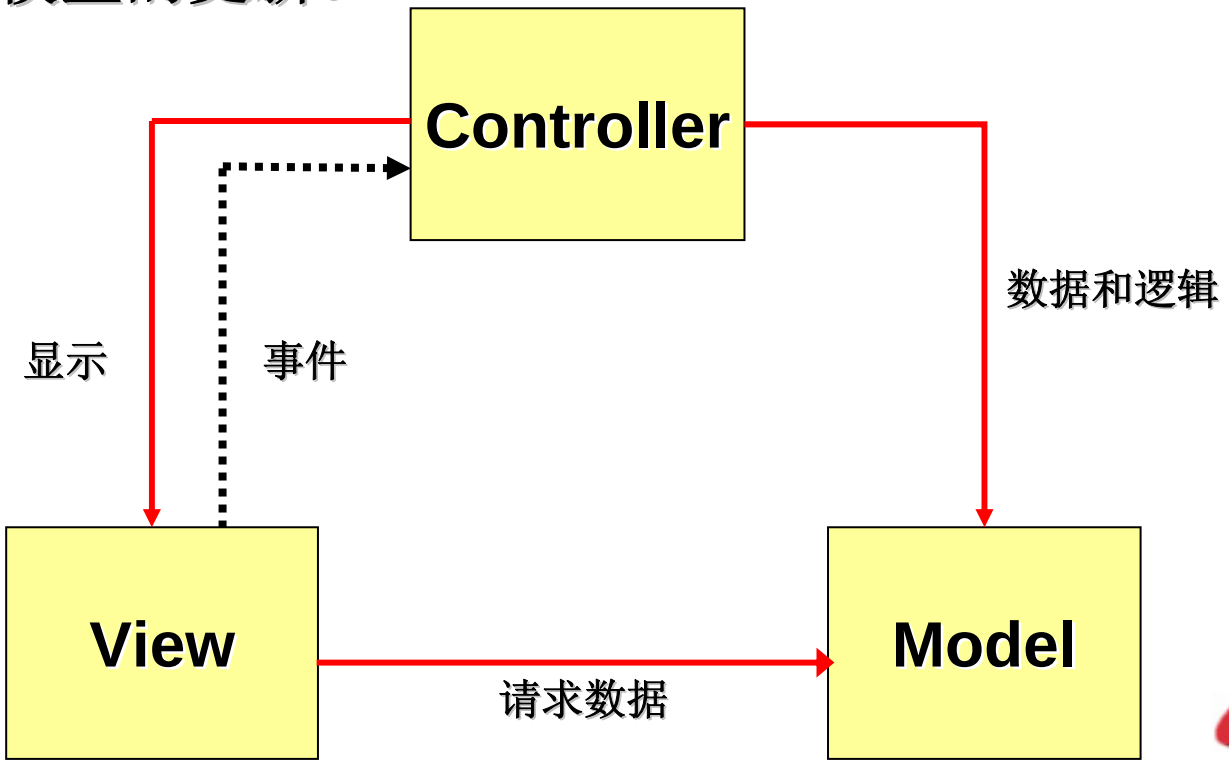


DTO: Data Transfer Object
DAO: Data Access Object

MVC 架构风格

MVC架构将应用分成3层:

- 模型实现从EIS中检索和更新信息。封装应用的状态和行为。
- 视图实现UI逻辑。渲染模型。
- 控制器控制数据模型和界面之间的交互。处理用户事件并且驱动视图和模型的更新。



在MVC中的模型层

- 模型封装应用的状态。
- 包含应用的业务逻辑的核心。
- 提供检索或者设置数据的API。
- 对应用的视图层和控制器层一无所知。
- 能够在具有不同的控制器和视图的应用中被重用。

在MVC中的视图层

- 视图支持应用的**UI**（或者**look-and-feel**）。
- 提供模型数据的表示。
- 能直接从模型中检索数据，但是不应该设置模型中的数据。
- 当在模型中发生变化时会被通知。

在MVC中的控制器层

- 控制器通过调用模型中合适的方法响应用户输入。
- 决定通过应用的用户路径。
- 创建和设置模型状态。

对象的重要组成

- 标识（名字）：类型
- 状态
- 行为

An Object

- **A discrete entity with a well-defined boundary and identity that encapsulates state and behavior; an instance of a class.**

讨论：

Java中Object类中的equals()和hashCode()（散列码）的作用。

对象的唯一性对容器类**HashSet**和**HashMap**的重要性。

面向对象的基本观点

① 客观世界是由对象组成的。

② 具有相同的数据和相同的操作的对象可以归并为一个类，对象是对象类的一个实例。从一个类可以产生许多对象。

③ 类可以派生出子类，子类继承父类的全部特征（属性和操作），并且又可以有自己的新特征。子类与父类形成类的层次结构。

④ 对象之间通过传递消息（事件）交互。

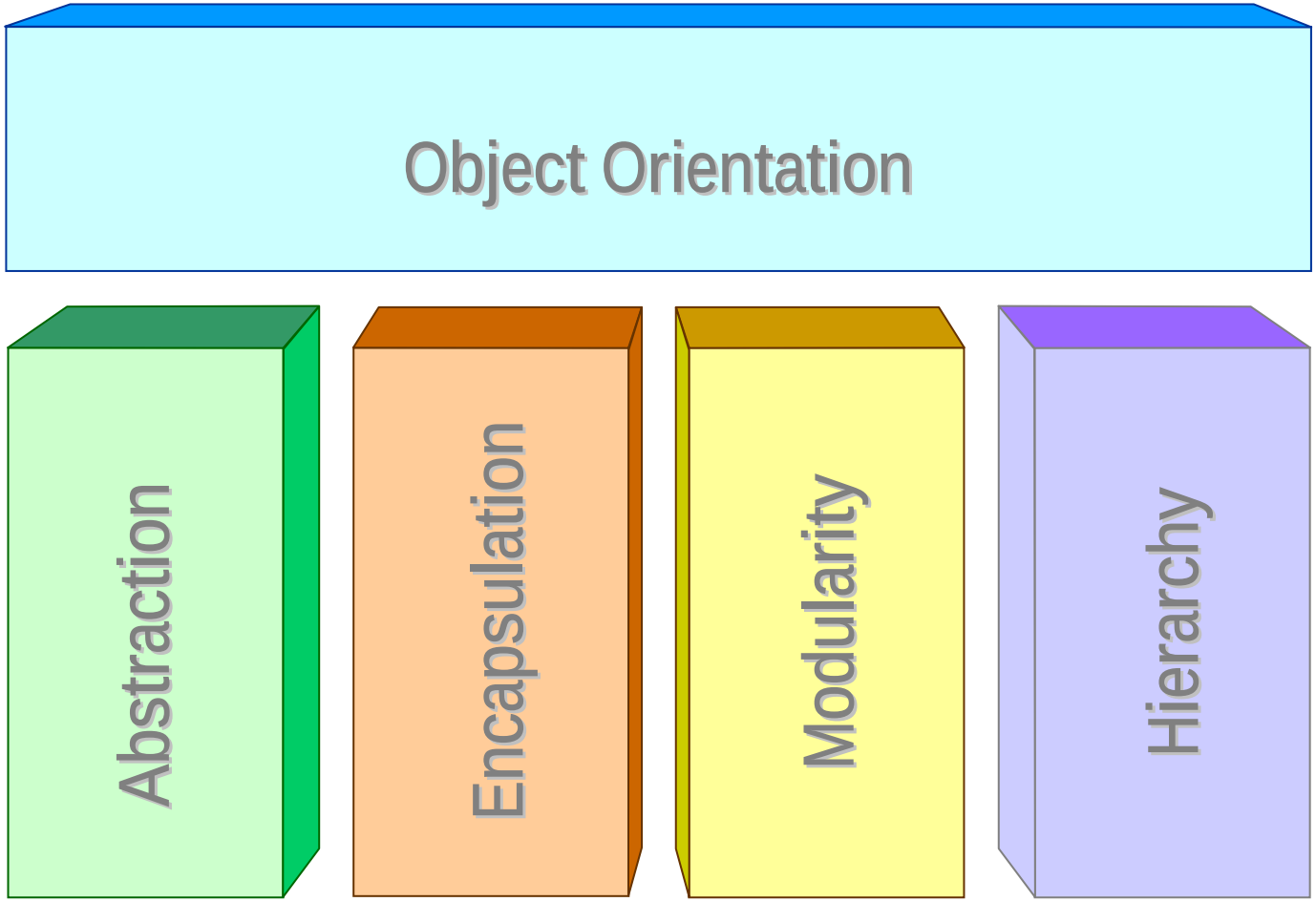
面向对象

面向对象 = 对象 (object)
+ 类 (classification)
+ 继承 (inheritance)
+ 通过消息的通信（交互）
(communication with messages)

(Peter Coad & Edward Yourdon)

如果一个计算机软件系统采用面向对象来建立模型并予以实现，它就是面向对象的系统。

Basic Principles of Object Orientation



面向对象的四条基本原理

- 抽象物（**Abstraction** ）
- 封装（**Encapsulation** ）
- 模块化（**Modularity**）
- 层次（**Hierarchy** ）

面向对象

类型　对象名 = new 具体类的构造器;

…

对象名•操作名（参数列表）；　（接受消息）

讨论题：

•讨论对象的**编译时类型**和**运行时类型**；

多态

- **Polymorphism allows the same message to be handled in different ways depending on the object that receives it.**
- 它为多个类定义了同样的操作
- 运行时多态是基于继承的
- 多态功能的实现是依赖于它所应用的对象
- **The essence of polymorphism is that instead of asking an object what type it is and then invoking some behaviours based on the answer，you just invoke the behaviour. The object，depending on its type，does the right thing.**

```
template <class T>        //C++
class A {
        typename  T  t;
   public:
    void mf() {  t.mf();}
};

A<Ai>  obj;     //  i=1，…，  n;
obj.mf();
```

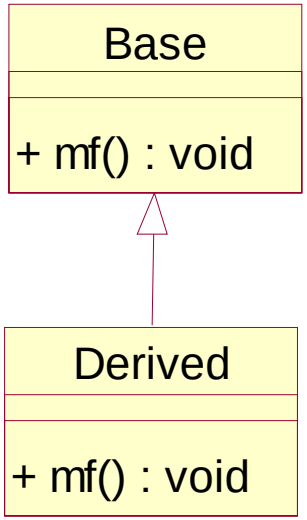
讨论：

继承和多态；

静态多态；

动态多态；

```
class Base {    //C++
public:
    void mf();   // non-virtual
};
class Derived : public Base {
public:
    void mf();
};
Base*  pBase  = new Derived();
pBase -> mf();
```

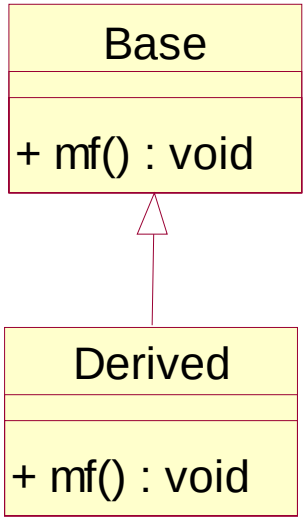


讨论：

继承和多态；

切片问题；

```
public class Base {   //Java
     public  void mf() {
         System.out.print("Base::mf")
     }
}
public class Derived extends Base {
     public  void mf() {
          System.out.print("Derived ::mf")
     }
}
```



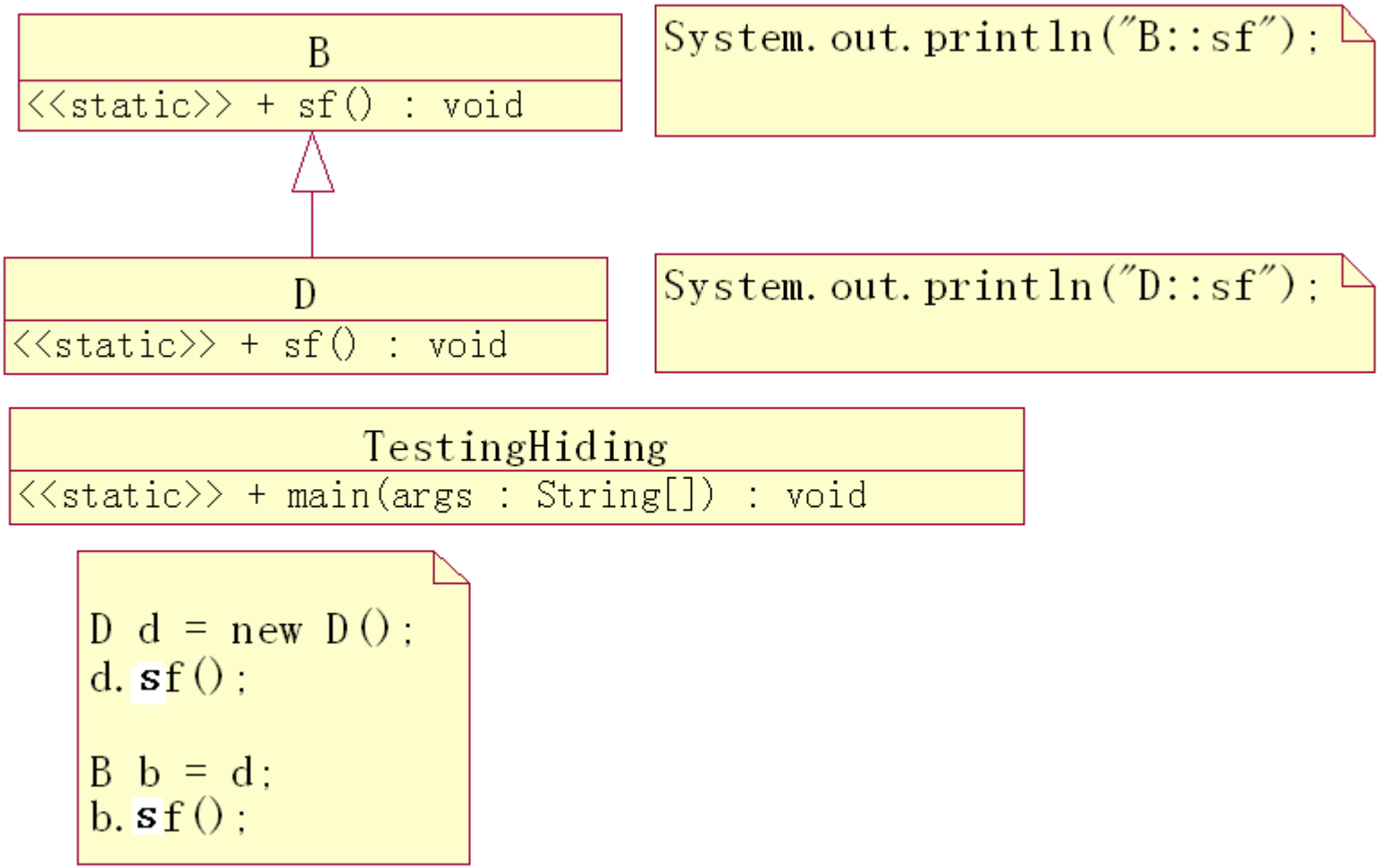

```
Base  base  = new Derived();
base. mf();
```

讨论：

继承和多态；

切片问题；

Redefining（重定义）



关键字this VS static

This
+ mf(a : int) : void <<static>> + mf(b : float) : void <<static>> + main(args : String[]) : void

```
This o = new This();

o.mf(1);
//equivalent to This.mf(This this = o, int a = 1)

o.mf(1.0f);
//equivalent to This.mf(float b = 1.0f)
```

讨论题：

- 静态绑定

IT服务中国

Scope
作用域/范围

神州数码
Digital China

Scope (作用域/范围）- Java

1. Block Scope （块作用域）
2. Method Scope （方法作用域）
3. Method -Prototype Scope (方法原型作用域)
4. Instance Block Scope
5. Static Block Scope
6. Instance Scope
7. Class Scope (类作用域）
8. Package Scope (包作用域）

局部变量

讨论题：

局部变量可以有哪些修饰符？

Scope (作用域/范围）- C++

1. Block Scope （块作用域）
2. Function Scope （函数作用域）
3. Function-Prototype Scope (函数原型作用域)
4. Class Scope (类作用域）
5. Package Scope (包作用域）
6. File-Static Scope （文件静态作用域）

局部变量

变量的分类

1. 类型
2. 范围
3. 内存位置
4. 生命周期

Java应用程序的内存结构

<table>
<tr><th>指 令</th><th colspan="3">数 据</th></tr>
<tr><td>Text Segment
（文本段）</td><td>Data Segment
（数据段）
静态存储区</td><td>Heap
（堆）</td><td>Stack
（栈）</td></tr>
</table>

思考和讨论题：

根据上图，实例化一个对象有几种方式？这几种方式之间有什么重大差别？

讨论Java程序的执行方式。

讨论不同内存位置上的变量的初始化问题。

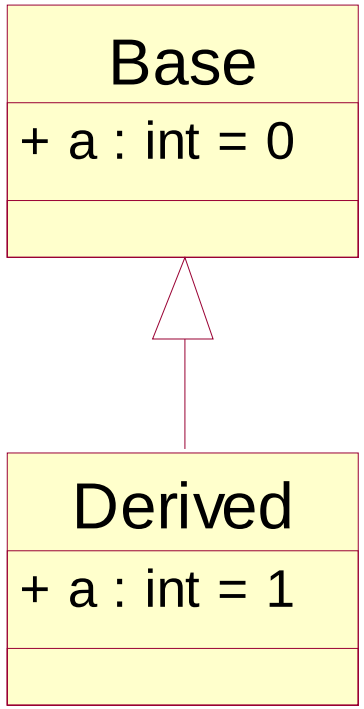
Java方法在文本段上的保存形式

1. 实例方法
   返回类型 包名.类名.方法名(类名 **this**, 参数列表)。
   调用形式：**obj.方法名(参数列表)**等价于
   类名.方法名(this = obj , 参数列表)
2. 类方法（静态方法）
   返回类型 包名.类名.方法名(参数列表)。
   调用形式：类名.方法名(参数列表)

Java对象的构造和初始化

1. 分层加载。将方法指令放在文本段上；将静态变量放在数据段上（同时进行缺省初始化），然后进行直接初始化，最后再调用静态块。对子类重复同样的动作。
2. 通过使用继承，为需要直接实例化的整个类型层次中的所有的属性在堆上一次性分配内存，并且对所有属性进行缺省初始化。
3. 分层构造（对每一层依次进行直接初始化，然后调用相应的构造器。）

```
public class Base {   //Java
      public   int  a = 0;
}
public class Derived  extends  Base {
      public   int  a = 1;
}
```




```
Base  base  = new Derived();
System.out.println(base.a);
```

讨论：

类型和作用域的关系；

Java对象成员的调用规则

B obj = new D();　　obj.成员；

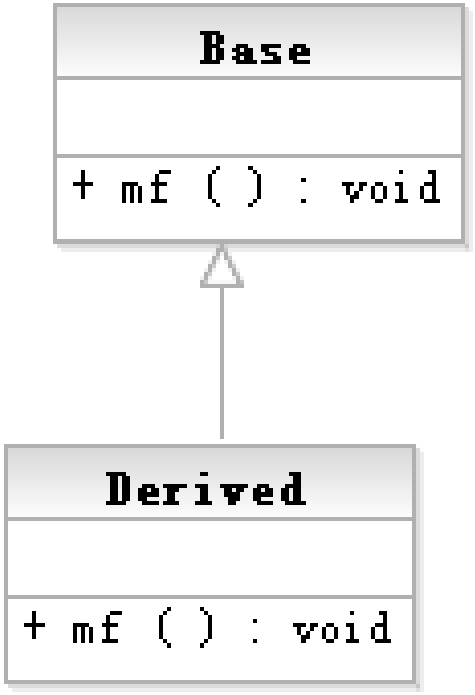
1. 如果成员是属性，它的取值由编译时类型决定。
2. 如果成员是方法，则要确定是进行静态绑定还是动态绑定；

a) 如果被调方法是非虚方法则进行静态绑定。被调方法由编译时类型决定。

b) 如果被调方法是虚方法则进行动态绑定。被调方法由运行时类型决定。

方法缺省是虚方法，但是私有的、静态的、最终的方法是非虚方法。抽象方法一定是虚方法。

Overriding and Visibility (1)




```
void mf() {
    System.out.println("Base::mf");
}
```

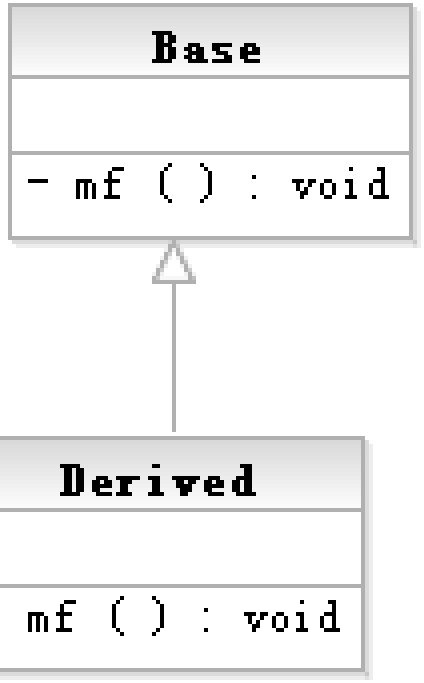
```
void mf() {
    System.out.println("Derived::mf");
}
```

```
Base b = new Derived();
b.mf();
```

输出：**Derived::mf**

Polymorphism

Overriding and Visibility (2)




```
void mf() {
    System.out.println("Base::mf");
}
```

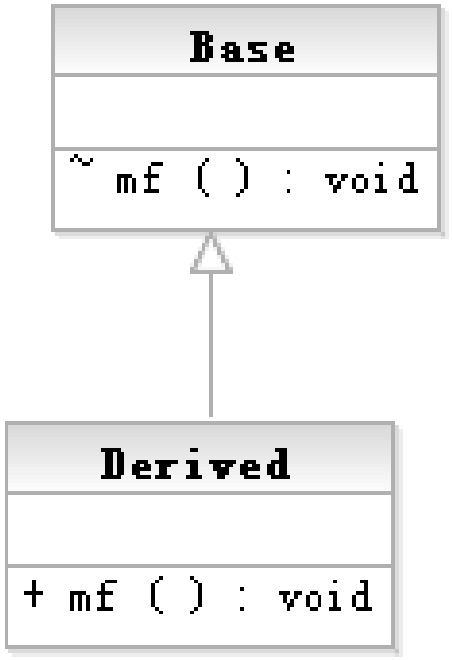
```
void mf() {
    System.out.println("Derived::mf");
}
```

```
Base b = new Derived();
b.mf();
```

输出：**Base::mf**

Bad Design !

Overriding and Visibility (3)




```
void mf() {
    System.out.println("Base::mf");
}
```

```
void mf() {
    System.out.println("Derived::mf");
}
```

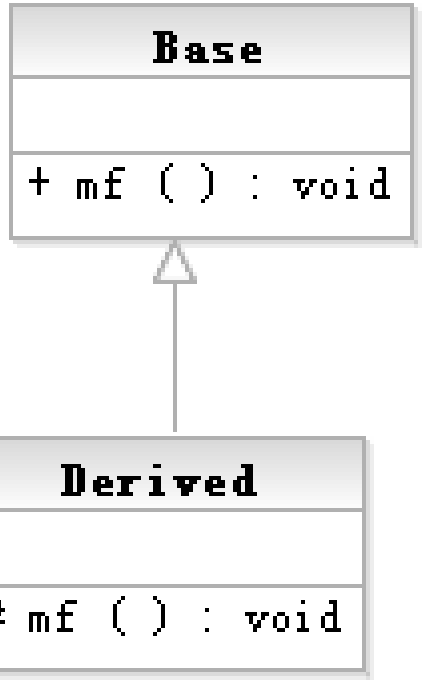
```
Base b = new Derived();
b.mf();
```

输出：**Derived::mf**

可见性可以扩大

Polymorphism

Overriding and Visibility (4)




```
void mf() {
    System.out.println("Base::mf");
}
```

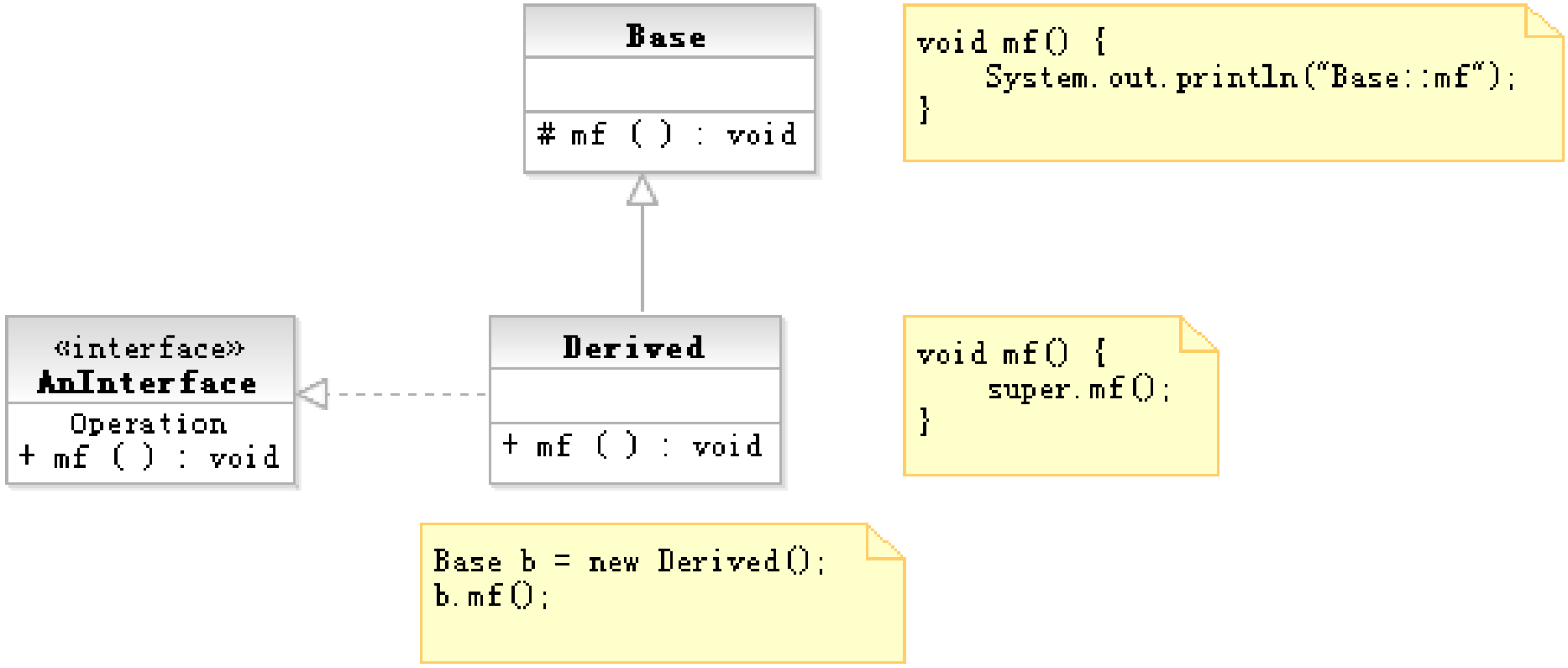
```
void mf() {
    System.out.println("Derived::mf");
}
```

```
Base b = new Derived();
b.mf();
```

语法错误！！

可见性不可以缩小！

Overriding and Visibility (5)



Overriding必须满足的条件

1. 覆盖只能够发生在继承树里面；
2. 被覆盖的方法一定是看得见的虚方法；
3. 覆盖方法的Signature和被覆盖方法的Signature一定要一致；
4. 被覆盖的方法的返回类型如果是基本类型，覆盖方法的返回类型和被覆盖方法的返回类型必须精确一致，不允许提升。如果返回类型是引用类型的，允许多态（JDK版本必须是1.5)。
5. 被覆盖的方法的可见性只能放大（可以在不同的包间被放大），不能缩小。
6. 满足异常规约；

Overloading VS Overriding

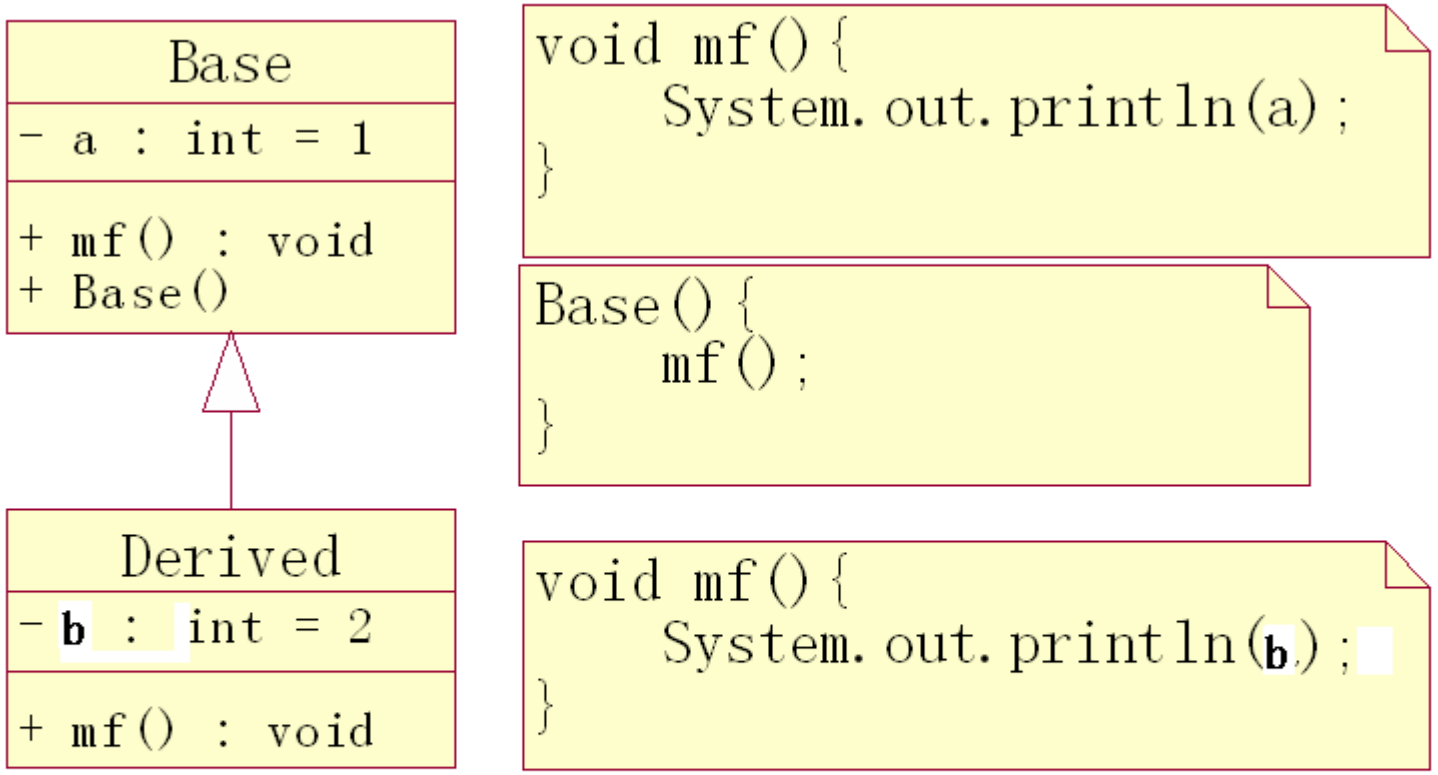
- Selection among overloaded methods is static, while selection among overridden methods is dynamic.
- The choice of which overloading to invoke is made at compile time.
- The correct version of an *overridden* method is chosen at run time, based on the run time type of the object on which the method is invoked.
- A safe, conservative policy is never to export two overloadings with the same number of parameters.

IT服务中国

Java多态专题

神州数码
Digital China

在构造器中调用虚方法（1）



1）Base obj= new Derived();

2）obj.mf();

在构造器中调用虚方法（2）

Base obj = new Derived();

(1)

a	b

(2)

0	0

Default Initialization

(3)

1	0

Explicit Initialization

(4) 调基类**Base**的构造器（在其中调了多态的**mf()**）

(5) 调派生类的 **mf()**打印**Derived**类中**b**的值

(6) 派生类初始化

1	2

IT服务中国

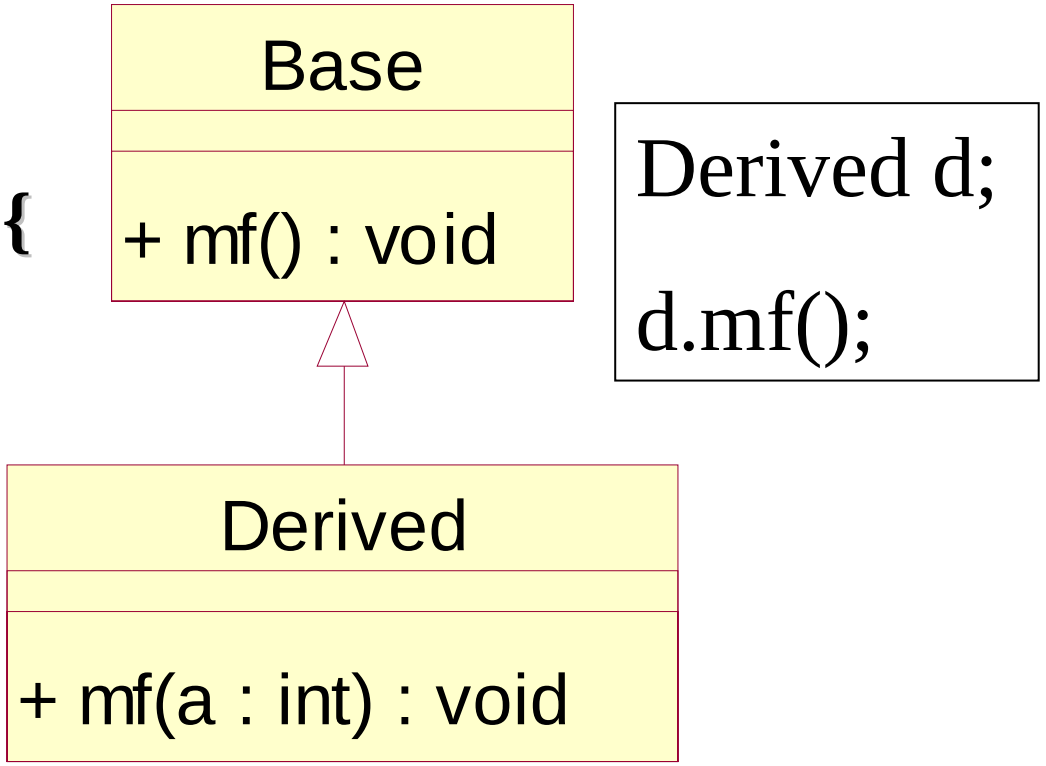
The Name Hiding Rule
(名字隐藏规则）

神州数码
Digital China

The Name Hiding Rule (C++)

内部作用域上的名字隐藏外部作用域上的名字。

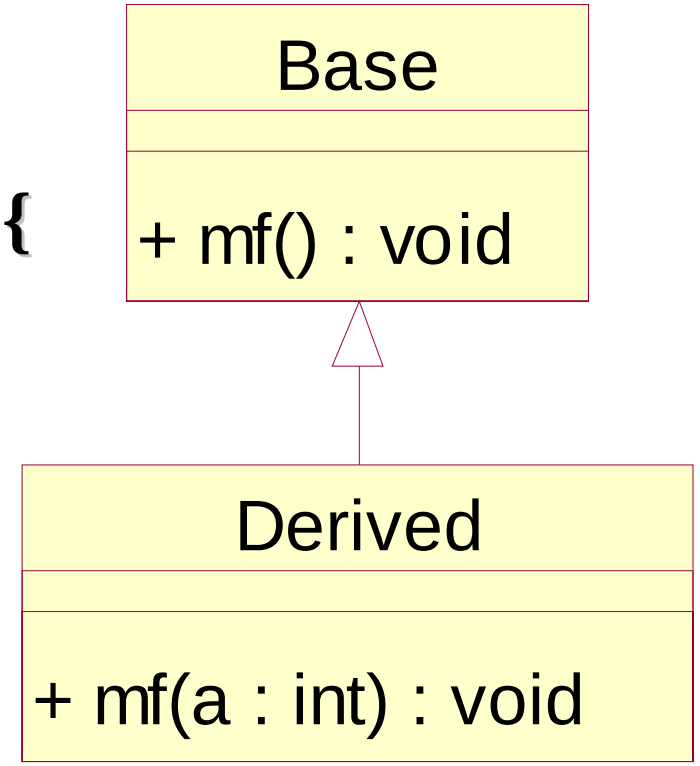
```
int a = 1;
int f(/*  int a ＝ 2  */) {
   int a =3;
    ….
   { ….
         int a =4 ;
   }
}
```



The Name Hiding Rule （1） - Java

内部作用域上的名字隐藏外部作用域上的名字。

```
int a = 1;
int f(/*  int a ＝ 2  */) {
   int a =3;
    ….
   { ….
//       int a =4 ;
   }
}
```




```
Derived d = new Derived ();
d.mf();
```

神州数码 Digital China

The Name Hiding Rule (C++)

内部作用域上的名字隐藏外部作用域上的名字。

Hide (隐藏）

Redefine （重定义）

Override （覆盖）

Overload （重载） //必须在同一作用域

The Name Hiding Rule (Java/C#)

内部作用域上的名字隐藏外部作用域上的名字。

Hide (隐藏）

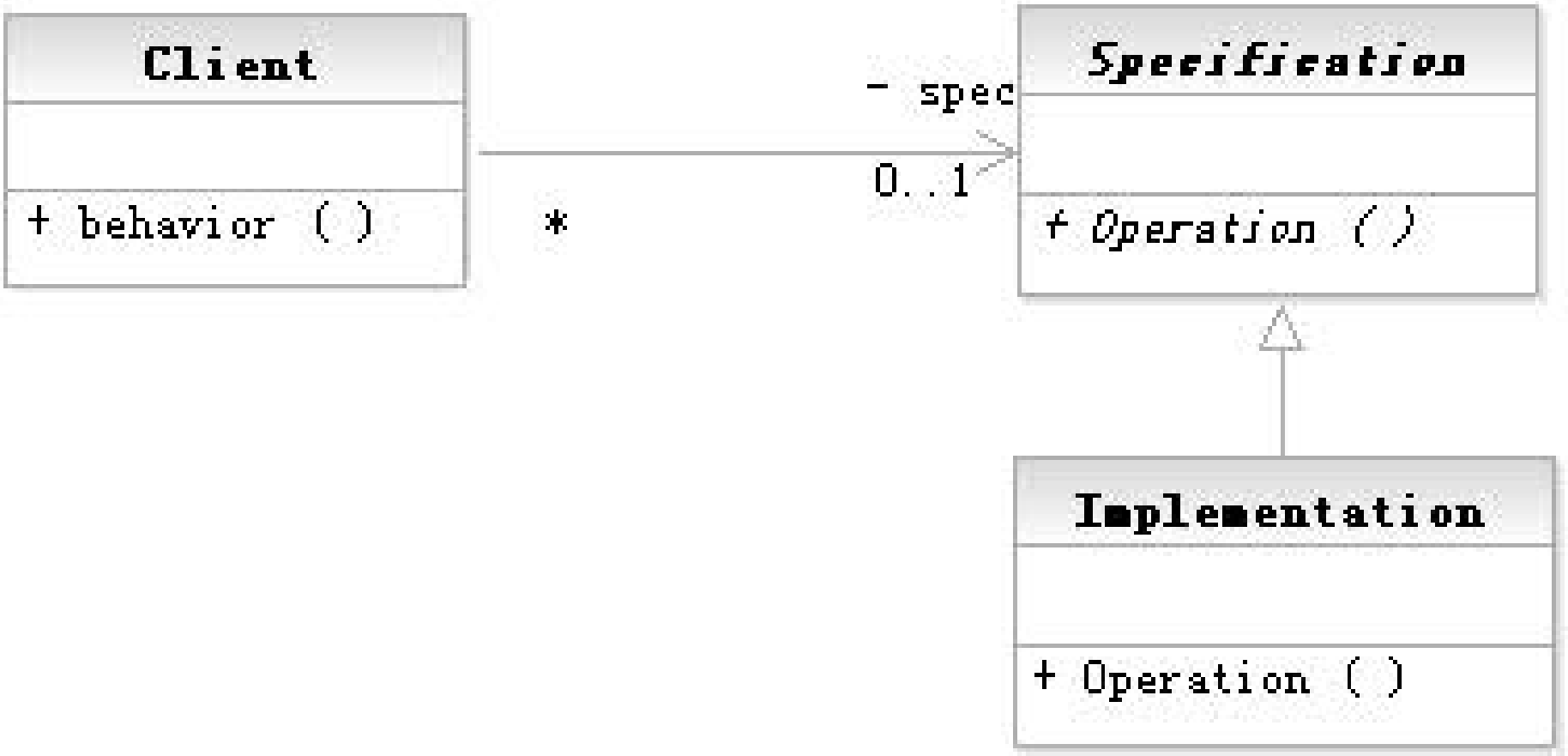
Redefine （重定义）

Override （覆盖）

Overload （重载） //可以在不同作用域

IT服务中

面向对象的核心概念模型



神州数码 Digital China

面向对象的核心概念模型