



深圳大学
Shenzhen University

第9讲

单例模式&适配器模式

软件体系结构与设计模式

Software Architecture & Design Pattern

深圳大学计算机与软件学院

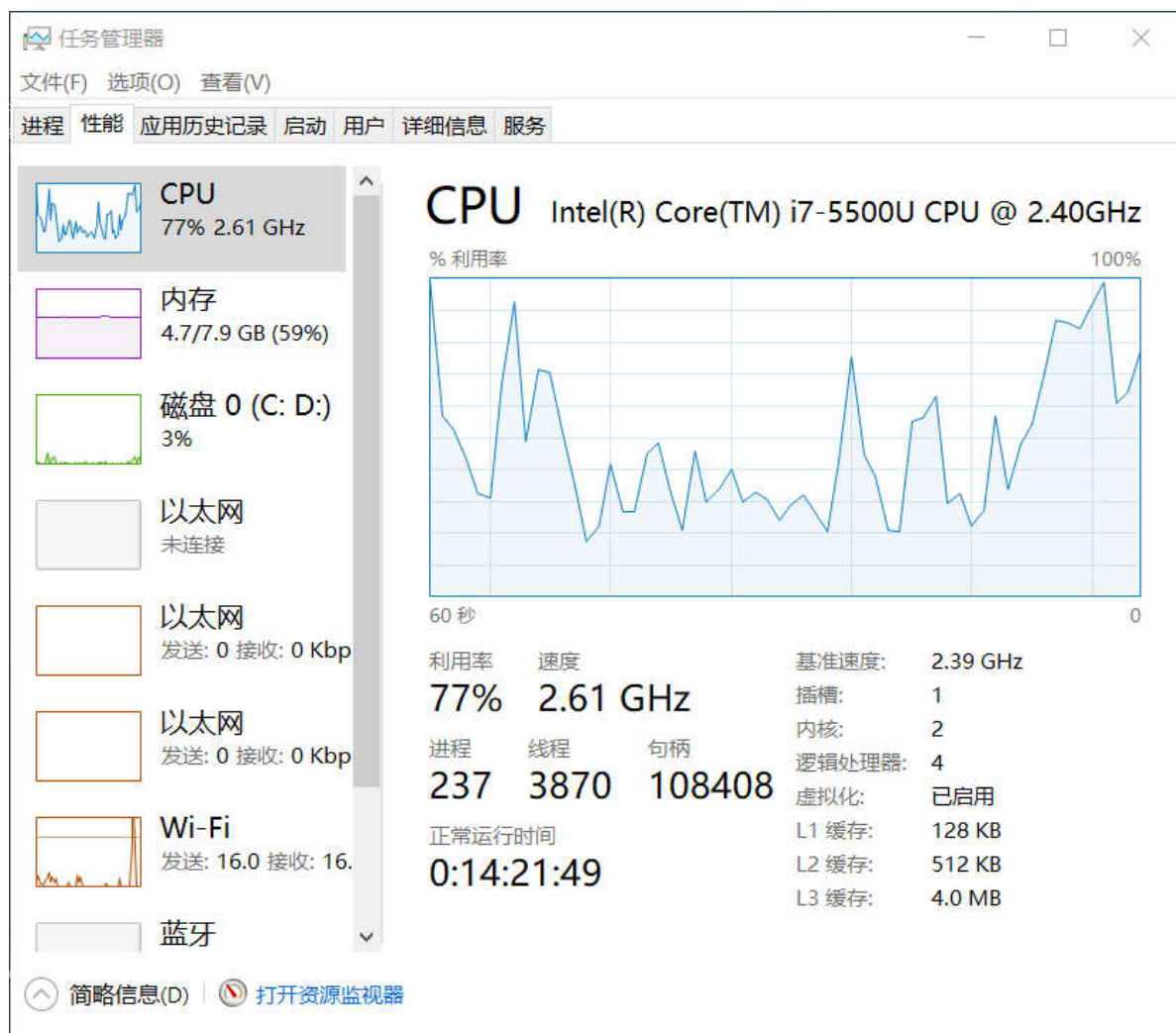


主要内容

- ◆ 9.1 单例模式
- ◆ 9.2 结构型模式概述
- ◆ 9.3 适配器模式

9.1 单例模式

■ 单例模式的模式动机



Windows , 在正常情况下只能打开唯一一个任务管理器 !



单例模式动机

■ 设计思路

- 如何保证一个类只有一个实例并且这个实例易于被访问？
 - 使用全局变量：可以确保对象随时可以被访问，但不能防止创建多个对象
 - 让类自身负责创建和保存它的唯一实例，并保证不能通过其他方法创建实例，同时提供一个访问该实例的方法

单例模式



单例模式定义

■ 单例模式的定义

单例模式：确保一个类**只有一个实例**，并提供一个**全局访问点**来访问这个唯一实例。

Singleton Pattern: Ensure a class has **only one instance**, and provide **a global point** of access to it.

□ 对象创建型模式



单例模式定义

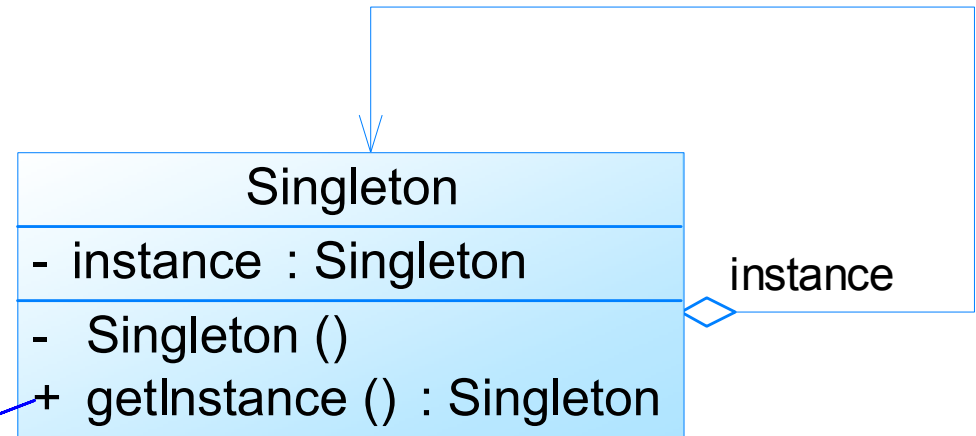
■ 要点

- 某个类只能有一个实例
- 必须自行创建这个实例
- 必须自行向整个系统提供这个实例

单例模式结构与实现

■ 单例模式结构

- 只包含一个单例角色：
- Singleton (单例)



```
if(instance==null)
    instance=new Singleton();
return instance;
```

单例模式结构与实现

■ 单例模式实现要点

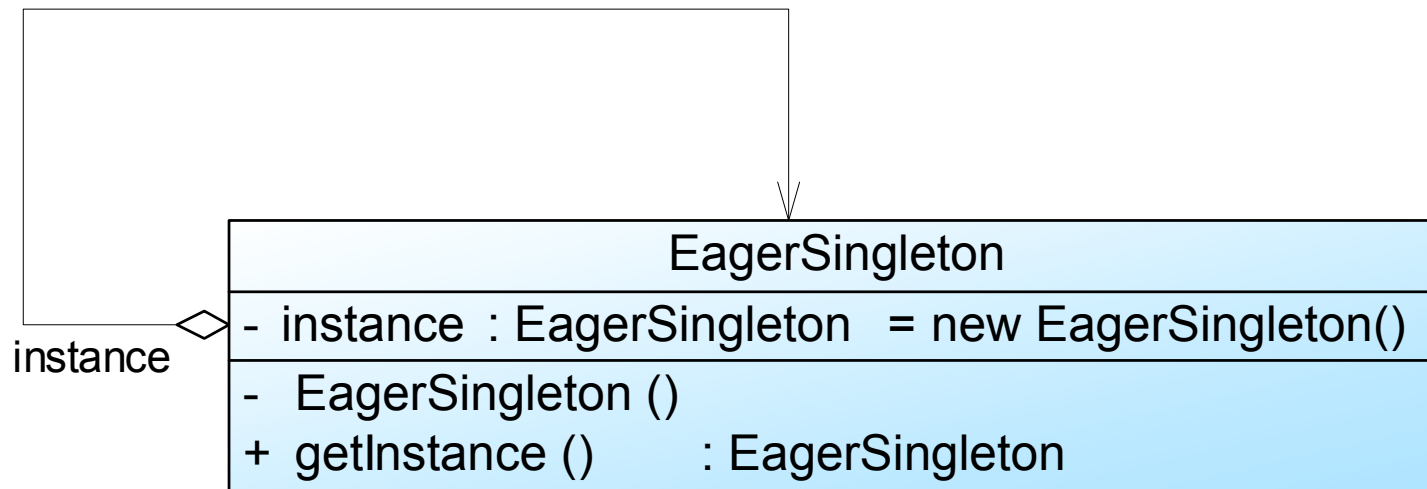
- 私有构造函数
- 静态私有成员变量（自身类型）
- 静态公有的工厂方法

■ 核心实现代码

```
public class Singleton {  
    //静态私有成员变量  
    private static Singleton instance=null;  
  
    //私有构造函数  
    private Singleton() {  
    }  
  
    //静态公有工厂方法，返回唯一实例  
    public static Singleton getInstance() {  
        if(instance==null)  
            instance=new Singleton();  
        return instance;  
    }  
}
```


单例模式分析

■ 饿汉式单例类(Eager Singleton)

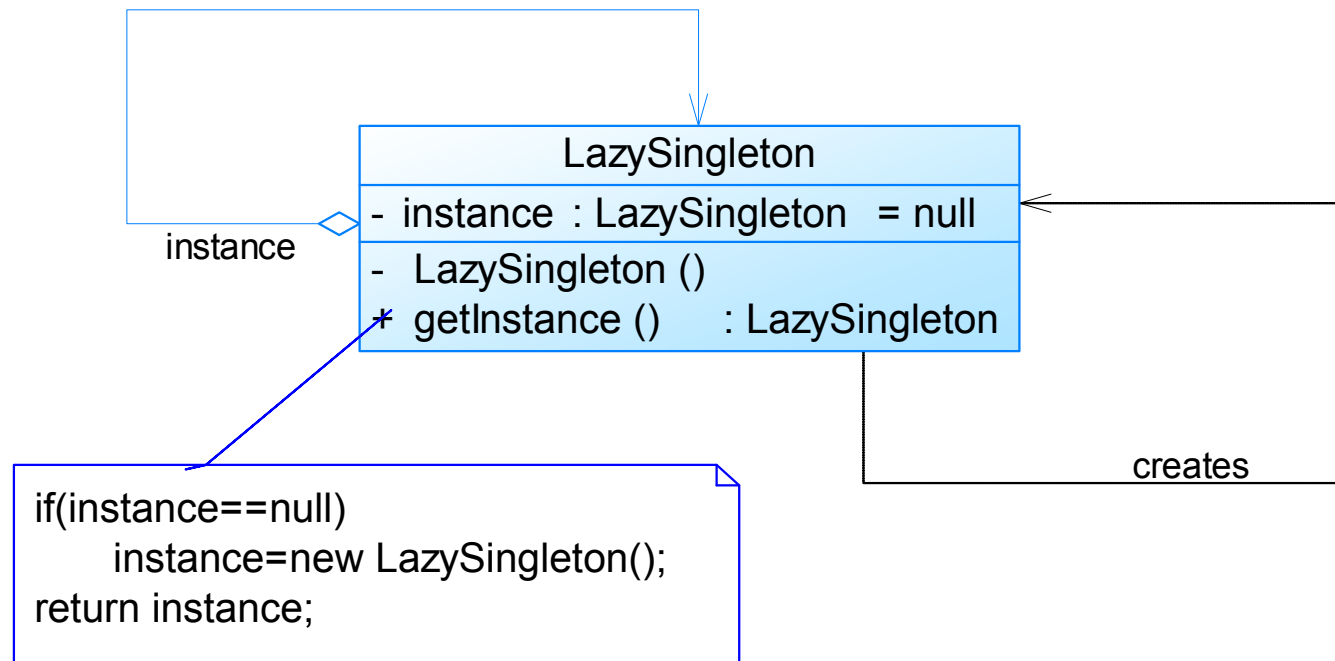


```
public class EagerSingleton {  
    private static final EagerSingleton instance = new EagerSingleton();  
    private EagerSingleton() { }  
  
    public static EagerSingleton getInstance() {  
        return instance;  
    }  
}
```

单例模式分析

■ 懒汉式单例类与双重检查锁定

□ 懒汉式单例类(Lazy Singleton)



单例模式分析

■ 懒汉式单例类与双重检查锁定

□ 延迟加载

```
public class LazySingleton {  
    private static LazySingleton instance = null;  
  
    private LazySingleton() { }  
  
    public static LazySingleton getInstance() {  
        if (instance == null) {  
            instance = new LazySingleton();  
        }  
        return instance;  
    }  
}
```

多个线程同时访问将导致创建多个单例对象！怎么办？

需要较长时间

单例模式分析

■ 懒汉式单例类与双重检查锁定

□ 延迟加载

```
public class LazySingleton {  
    private static LazySingleton instance = null;  
  
    private LazySingleton() { }  
  
    synchronized public static LazySingleton getInstance() {  
        if (instance == null) {  
            instance = new LazySingleton();  
        }  
        return instance;  
    }  
}
```

锁方法



单例模式分析

■ 懒汉式单例类与双重检查锁定

□ 延迟加载

锁代码段

```
.....  
public static LazySingleton getInstance() {  
    if (instance == null) {  
        synchronized (LazySingleton.class) {  
            instance = new LazySingleton();  
        }  
    }  
    return instance;  
}  
.....
```

单例模式分析

■ 懒汉式单例类与双重检查锁定

□ 延迟加载

```
public class LazySingleton {  
    private volatile static LazySingleton instance = null;  
  
    private LazySingleton() { }  
  
    public static LazySingleton getInstance() {  
        //第一重判断  
        if (instance == null) {  
            //锁定代码块  
            synchronized (LazySingleton.class) {  
                //第二重判断  
                if (instance == null) {  
                    instance = new LazySingleton(); //创建单例实例  
                }  
            }  
        }  
        return instance;  
    }  
}
```

Double-Check
Locking
双重检查锁定





单例模式分析

■ 饿汉式单例类与懒汉式单例类的比较

- **饿汉式单例类**：无须考虑多个线程同时访问的问题；调用速度和反应时间优于懒汉式单例；资源利用效率不及懒汉式单例；系统加载时间可能会比较长
- **懒汉式单例类**：实现了延迟加载；必须处理好多个线程同时访问的问题；需通过双重检查锁定等机制进行控制，将导致系统性能受到一定影响



单例模式分析

- 使用静态内部类实现单例模式
 - Java语言中最好的实现方式
 - Initialization on Demand Holder (IoDH): 使用静态内部类(static inner class)

//Initialization on Demand Holder

```
public class Singleton {  
    private Singleton() {  
    }  
}
```

//静态内部类

```
private static class HolderClass {  
    private final static Singleton instance = new Singleton();  
}
```

```
public static Singleton getInstance() {  
    return HolderClass.instance;  
}
```

```
public static void main(String args[]) {  
    Singleton s1, s2;  
    s1 = Singleton.getInstance();  
    s2 = Singleton.getInstance();  
    System.out.println(s1==s2);  
}
```

```
}
```



单例模式实例与解析

■ 单例模式实例

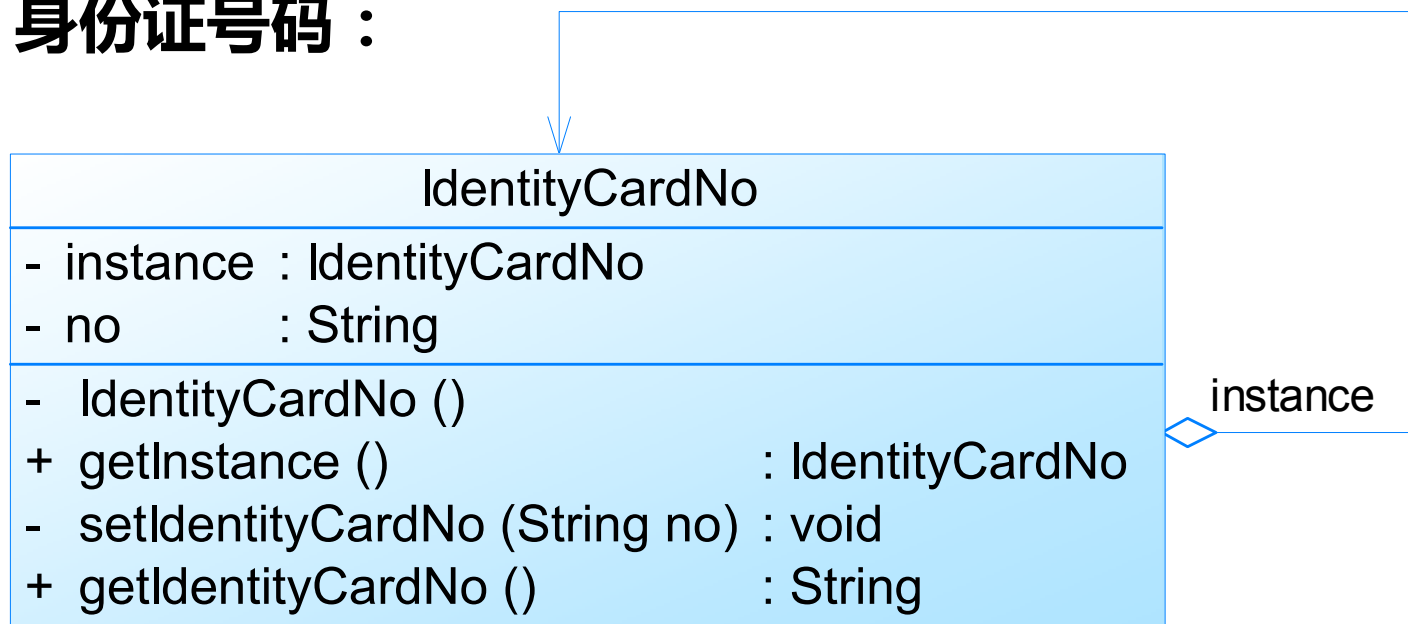
□ 身份证号码：实例说明

- 在现实生活中，居民身份证号码具有唯一性，同一个人不允许有多个身份证号码，第一次申请身份证时将给居民分配一个身份证号码，如果之后因为遗失等原因补办时，还是使用原来的身份证号码，不会产生新的号码。现使用单例模式模拟该场景。

单例模式实例与解析

■ 单例模式实例

□ 身份证号码：



□ 参考代码Singleton\sample01

IdentityCardNo.java

```
1 public class IdentityCardNo
2 {
3     private static IdentityCardNo instance=null;
4     private String no;
5
6     private IdentityCardNo()
7     {
8     }
9
10    public static IdentityCardNo getInstance()
11    {
12        if(instance==null)
13        {
14            System.out.println("第一次办理身份证，分配新号码！");
15            instance=new IdentityCardNo();
16            instance.setIdentityCardNo("No400011112222");
17        }
18        else
19        {
20            System.out.println("重复办理身份证，获取旧号码！");
21        }
22        return instance;
23    }
```

IdentityCardNo.java

```
24
25    private void setIdentityCardNo(String no)
26    {
27        this.no=no;
28    }
29
30    public String getIdentityCardNo()
31    {
32        return this.no;
33    }
34
35 }
```

```
1 public class Client
2 {
3     public static void main(String a[])
4     {
5         IdentityCardNo no1,no2;
6         no1=IdentityCardNo.getInstance();
7         no2=IdentityCardNo.getInstance();
8         System.out.println("身份证号码是否一致: " + (no1==no2));
9
10        String str1,str2;
11        str1=no1.getIdentityCardNo();
12        str2=no1.getIdentityCardNo();
13        System.out.println("第一次号码: " + str1);
14        System.out.println("第二次号码: " + str2);
15        System.out.println("内容是否相等: " + str1.equalsIgnoreCase(str2));
16        System.out.println("是否是相同对象: " + (str1==str2));
17    }
18 }
```



单例模式实例与解析

■ 单例模式实例

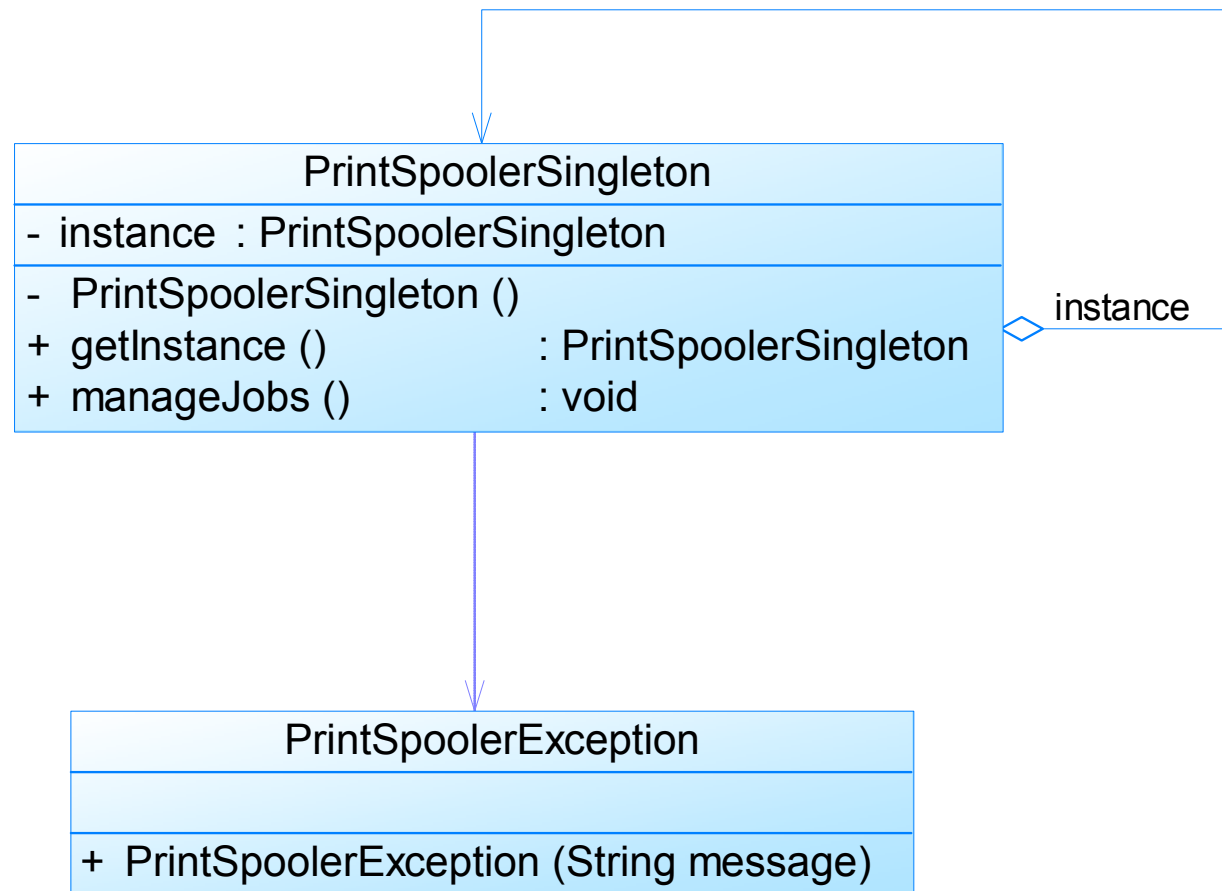
□ 打印池：实例说明

- 在操作系统中，打印池(Print Spooler)是一个用于管理打印任务的应用程序，通过打印池用户可以删除、中止或者改变打印任务的优先级，在一个系统中只允许运行一个打印池对象，如果重复创建打印池则抛出异常。现使用单例模式来模拟实现打印池的设计。

单例模式实例与解析

■ 单例模式实例

□ 打印池：参考类图



单例模式实例与解析

■ 单例模式实例

- 打印池：参考代码
- DesignPatterns之singleton包

```
PrintSpoolerException.java ✖
1 package singleton;
2
3 public class PrintSpoolerException extends Exception
4 {
5     public PrintSpoolerException(String message)
6     {
7         super(message);
8     }
9 }
```



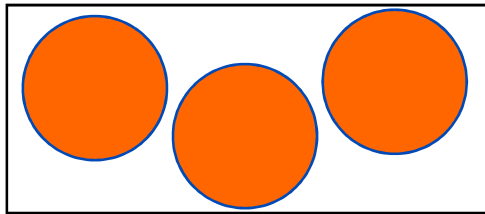
```
3 public class PrintSpoolerSingleton
4 {
5     private static PrintSpoolerSingleton instance=null;
6
7     private PrintSpoolerSingleton()
8     {
9     }
10
11     public static PrintSpoolerSingleton getInstance() throws PrintSpoolerException
12     {
13         if(instance==null)
14         {
15             System.out.println("创建打印池!");
16             instance=new PrintSpoolerSingleton();
17         }
18         else
19         {
20             throw new PrintSpoolerException("打印池正在工作中!");
21         }
22         return instance;
23     }
24
25     public void manageJobs()
26     {
27         System.out.println("管理打印任务!");
28     }
29 }
```

```
2
3 public class Client
4 {
5     public static void main(String a[])
6     {
7         PrintSpoolerSingleton ps1,ps2;
8         try
9         {
10             ps1=PrintSpoolerSingleton.getInstance();
11             ps1.manageJobs();
12         }
13         catch(PrintSpoolerException e)
14         {
15             System.out.println(e.getMessage());
16         }
17         System.out.println("-----");
18         try
19         {
20             ps2=PrintSpoolerSingleton.getInstance();
21             ps2.manageJobs();
22         }
23         catch(PrintSpoolerException e)
24         {
25             System.out.println(e.getMessage());
26         }
27     }
28 }
```

单例模式效果与应用

■ 单例模式优点：

- 提供了对唯一实例的受控访问
- 可以节约系统资源，提高系统的性能
- 允许可变数目的实例（多例类）



Multiton
- array : Multiton[]
- Multiton ()
+ getInstance () : Multiton
+ random () : int



单例模式效果与应用

■ 单例模式缺点：

- 扩展困难（缺少抽象层）
- 单例类的职责过重
- 由于自动垃圾回收机制，可能会导致共享的单例对象的状态丢失



单例模式效果与应用

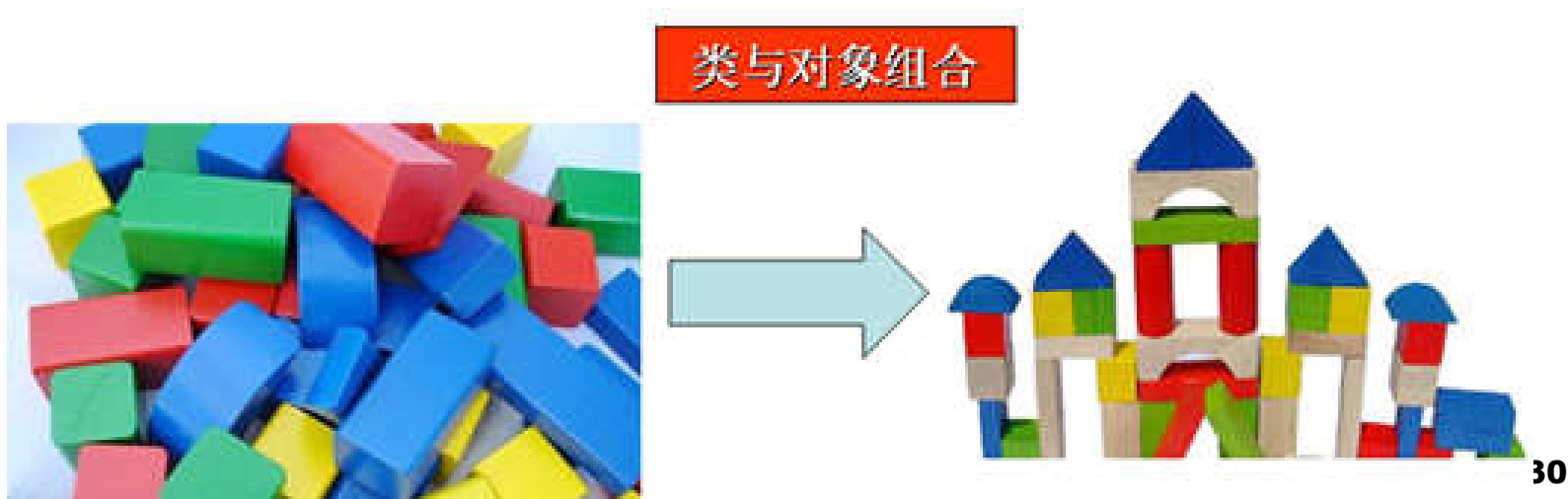
■ 在以下情况下可以使用单例模式：

- 系统只需要一个实例对象，或者因为资源消耗太大而只允许创建一个对象
- 客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例

9.2 结构型模式概述

■ 结构型模式

- **结构型模式(Structural Pattern)**关注如何将现有类或对象组织在一起形成更加强大的结构
- 不同的结构型模式从不同的角度组合类或对象，它们在尽可能满足各种面向对象设计原则的同时为类或对象的组合提供一系列巧妙的解决方案



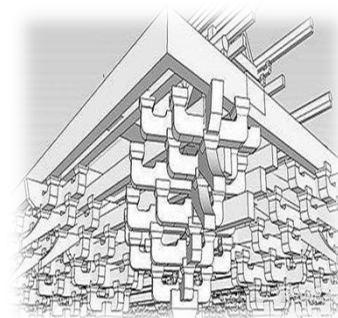
结构型模式分类

■ 类结构型模式

- 关心类的组合，由多个类组合成一个更大的系统，在类结构型模式中一般只存在继承关系和实现关系

■ 对象结构型模式

- 关心类与对象的组合，通过关联关系，在一个类中定义另一个类的实例对象，然后通过该对象调用相应的方法





结构型模式一览表

模式名称	定义	学习难度	使用频率
适配器模式 (Adapter Pattern)	将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作。	★★☆☆☆	★★★★☆
桥接模式 (Bridge Pattern)	将抽象部分与它的实现部分解耦，使得两者都能够独立变化。	★★★★☆	★★★★☆
组合模式 (Composite Pattern)	组合多个对象形成树形结构，以表示具有部分-整体关系的层次结构。组合模式让客户端可以统一对待单个对象和组合对象。	★★★★☆	★★★★☆
装饰模式 (Decorator Pattern)	动态地给一个对象增加一些额外的职责。就扩展功能而言，装饰模式提供了一种比使用子类更加灵活的替代方案。	★★★★☆	★★★★☆
外观模式 (Facade Pattern)	为子系统中的一组接口提供一个统一的入口。外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。	★☆☆☆☆	★★★★★
享元模式 (Flyweight Pattern)	运用共享技术有效地支持大量细粒度对象的复用。	★★★★☆	★☆☆☆☆
代理模式 (Proxy Pattern)	给某一个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。	★★★★☆	★★★★☆

9.3 适配器模式

■ 适配器模式动机

□ 电源适配器



□ 现实生活：

- 不兼容：生活用电220V \leftrightarrow 笔记本电脑20V
- 引入 AC Adapter (交流电适配器)

□ 软件开发：

- 存在不兼容的结构，例如方法名不一致
- 引入适配器模式



适配器模式定义

□ 适配器模式的定义

适配器模式：将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作。

Adapter Pattern: Convert the interface of a class into another interface clients expect. Adapter lets classes **work together** that couldn't otherwise because of **incompatible** interfaces.

□ 对象结构型模式 / 类结构型模式

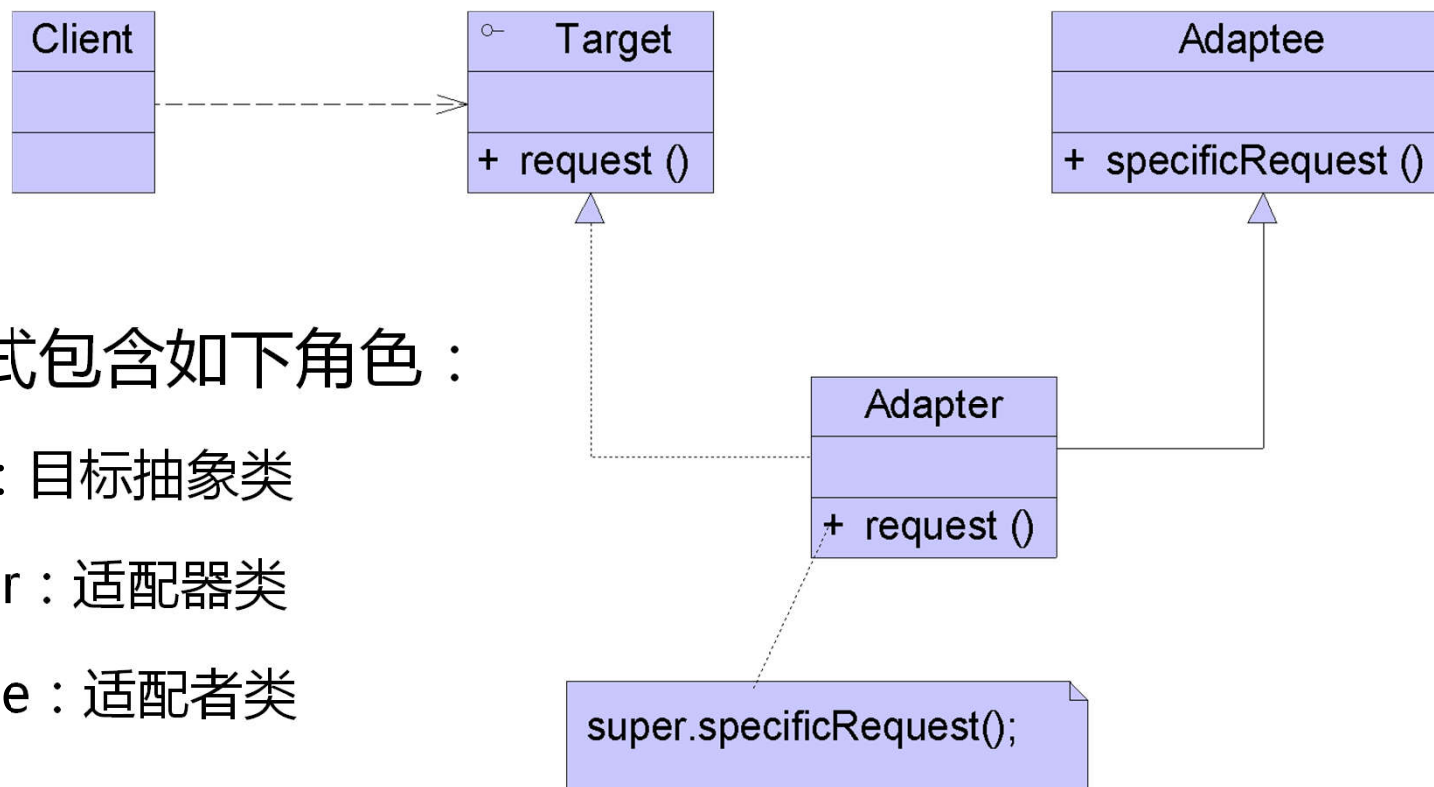
□ 别名为包装器(Wrapper)模式

□ 定义中所提及的接口是指广义的接口，它可以表示一个方法或者方法的集合

适配器模式结构与分析

■ 适配器模式结构

□ 类适配器

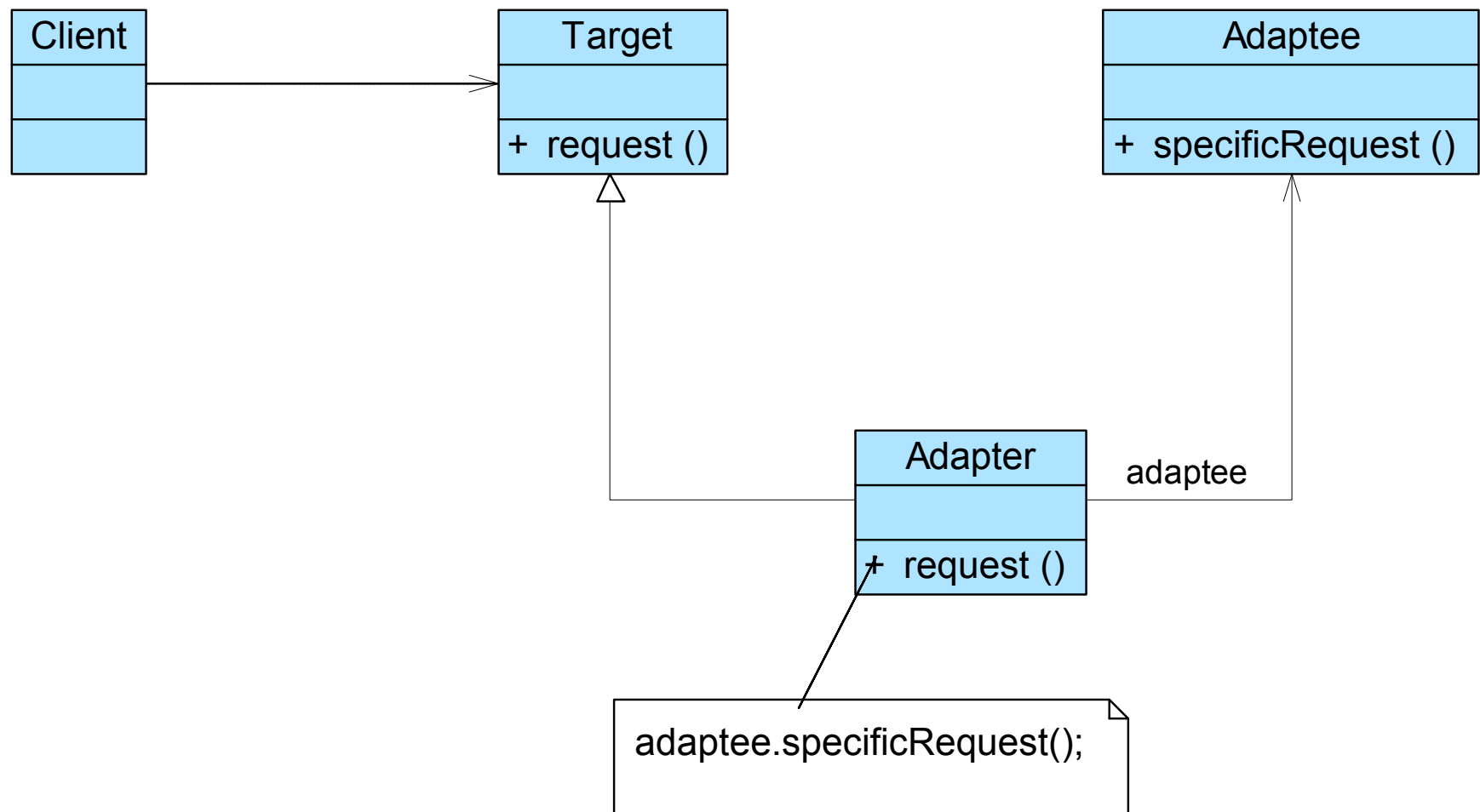


□ 适配器模式包含如下角色：

- Target：目标抽象类
- Adapter：适配器类
- Adaptee：适配者类

适配器模式结构与分析

- 适配器模式结构
 - 对象适配器





适配器模式分析

- 典型的类适配器示例代码：

```
public class Adapter extends Adaptee implements Target {  
    public void request() {  
        super.specificRequest();  
    }  
}
```

- 典型的对象适配器示例代码：

```
public class Adapter extends Target {  
    private Adaptee adaptee; //维持一个对适配者对象的引用  
    public Adapter(Adaptee adaptee) {  
        this.adaptee=adaptee;  
    }  
  
    public void request() {  
        adaptee.specificRequest(); //转发调用  
    }  
}
```



适配器模式实例与解析

■ 适配器模式实例

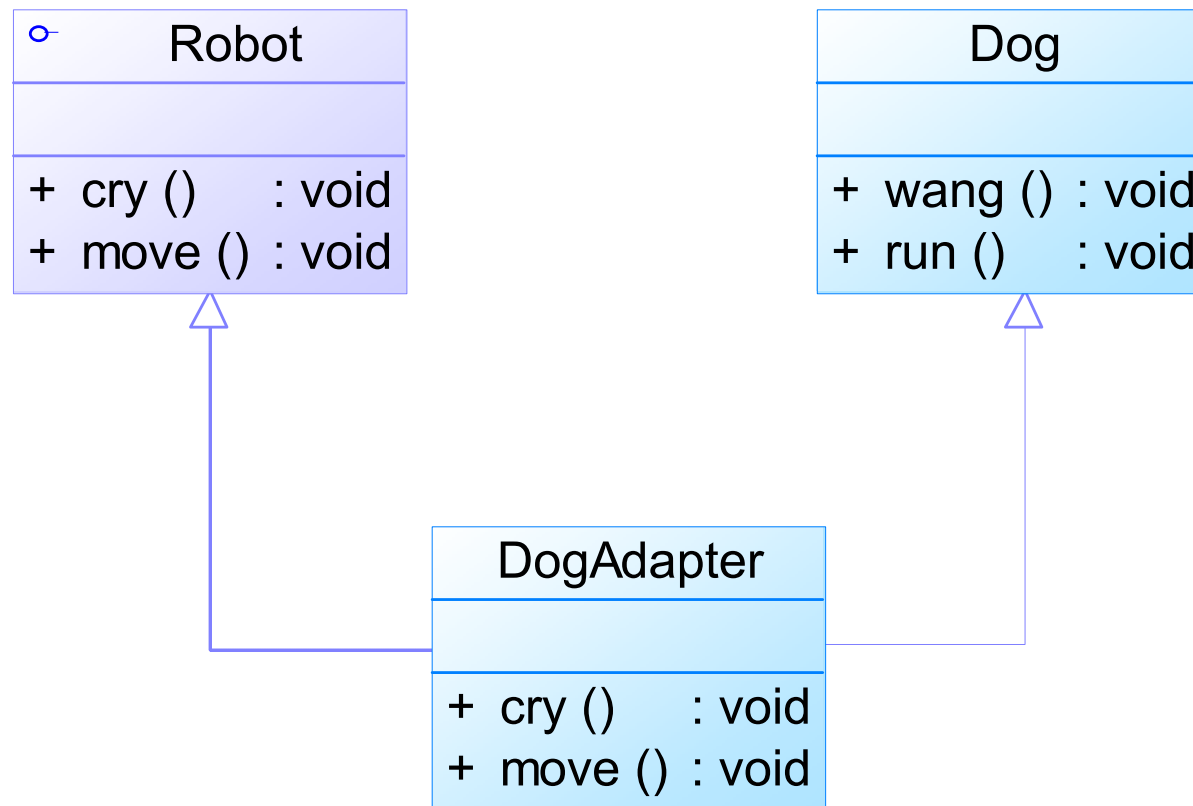
□ 仿生机器人：实例说明

- 现需要设计一个可以模拟各种动物行为的机器人，在机器人中定义了一系列方法，如机器人叫喊方法cry()、机器人移动方法move()等。如果希望在不修改已有代码的基础上使得机器人能够像狗一样叫，像狗一样跑，使用适配器模式进行系统设计。

适配器模式实例与解析

■ 适配器模式实例

□ 仿生机器人：参考类图



适配器模式实例与解析

■ 适配器模式实例

□ 仿生机器人：参考代码

■ DesignPatterns之adapter包

```
Robot.java
1 package adapter;
2
3 public interface Robot
4 {
5     public void cry();
6     public void move();
7 }
```

```
Dog.java
1 package adapter;
2
3 public class Dog
4 {
5     public void wang()
6     {
7         System.out.println("狗汪汪叫!");
8     }
9
10    public void run()
11    {
12        System.out.println("狗快快跑!");
13    }
14 }
```


DogAdapter.java

```
1 package adapter;
2
3 public class DogAdapter extends Dog implements Robot
4 {
5     public void cry()
6     {
7         System.out.print("机器人模仿: ");
8         super.wang();
9     }
10
11     public void move()
12     {
13         System.out.print("机器人模仿: ");
14         super.run();
15     }
16 }
```

Adapterconfig.xml

```
1 <?xml version="1.0"?>
2 <config>
3     <className>adapter.DogAdapter</className>
4 </config>
```

XMLUtil.java

```
3 import javax.xml.parsers.*;
4 import org.w3c.dom.*;
5 import org.xml.sax.SAXException;
6 import java.io.*;
7 public class XMLUtil
8 {
9     //该方法用于从XML配置文件中提取具体类名，并返回一个实例对象
10    public static Object getBean()
11    {
12        try
13        {
14            //创建文档对象
15            DocumentBuilderFactory dFactory = DocumentBuilderFactory.newInstance();
16            DocumentBuilder builder = dFactory.newDocumentBuilder();
17            Document doc;
18            doc = builder.parse(new File("Adapterconfig.xml"));
19
20            //获取包含类名的文本节点
21            NodeList nl = doc.getElementsByTagName("className");
22            Node classNode=nl.item(0).getFirstChild();
23            String cName=classNode.getNodeValue();
24            //通过类名生成实例对象并将其返回
25            Class c=Class.forName(cName);
26            Object obj=c.newInstance();
27            return obj;
28        }
```

XMLUtil.java

```
29         catch(Exception e)
30         {
31             e.printStackTrace();
32             return null;
33         }
34     }
35 }
```

```

Client.java ✕
1 package adapter;
2
3 public class Client
4 {
5     public static void main(String args[])
6     {
7         Robot robot=(Robot)XMLUtil.getBean();
8         robot.cry();
9         robot.move();
10    }
11 }

```

如果存在一个Bird类及其适配器类，可修改配置文件为BirdAdapter，使机器人具有鸟的行为。

<pre> Bird.java ✕ 3 public class Bird 4 { 5 public void tweedle() 6 { 7 System.out.println("鸟儿叽叽叫！"); 8 } 9 10 public void fly() 11 { 12 System.out.println("鸟儿快快飞！"); 13 } 14 } </pre>	<pre> BirdAdapter.java ✕ 3 public class BirdAdapter extends Bird implements Robot 4 { 5 public void cry() 6 { 7 System.out.print("机器人模仿："); 8 super.tweedle(); 9 } 10 11 public void move() 12 { 13 System.out.print("机器人模仿："); 14 super.fly(); 15 } 16 } </pre>
---	---



适配器模式实例与解析

■ 适配器模式实例

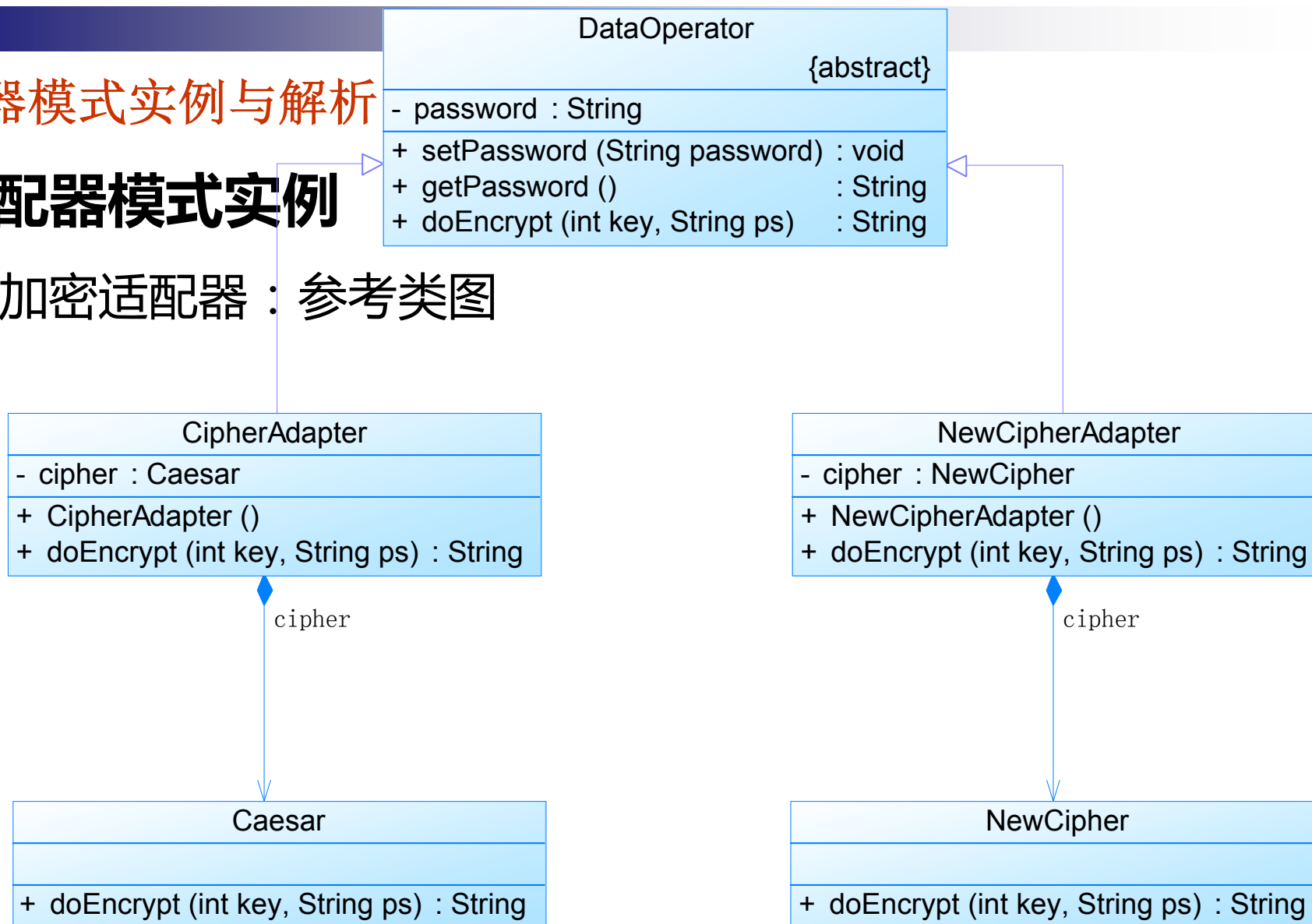
□ 加密适配器：实例说明

- 某系统需要提供一个加密模块，将用户信息（如密码等机密信息）加密之后再存储在数据库中，系统已经定义好了数据库操作类。为了提高开发效率，现需要重用已有的加密算法，这些算法封装在一些由第三方提供的类中，有些甚至没有源代码。使用适配器模式设计该加密模块，实现在不修改现有类的基础上重用第三方加密方法。

适配器模式实例与解析

■ 适配器模式实例

□ 加密适配器：参考类图



适配器模式实例与解析

■ 适配器模式实例

□ 加密适配器：参考代码

目标抽象类（数据操作类）

```
DataOperation.java ✕  
1 public abstract class DataOperation  
2 {  
3     private String password;  
4  
5     public void setPassword(String password)  
6     {  
7         this.password=password;  
8     }  
9  
10    public String getPassword()  
11    {  
12        return this.password;  
13    }  
14  
15    public abstract String doEncrypt(int key,String ps);  
16 }
```



```
1 public final class Caesar
2 {
3     public String doEncrypt(int key,String ps)
4     {
5         String es="";
6         for(int i=0;i<ps.length();i++)
7         {
8             char c=ps.charAt(i);
9             if(c>='a'&&c<='z')
10            {
11                c+=key%26;
12                if(c>'z') c-=26;
13                if(c<'a') c+=26;
14            }
15            if(c>='A'&&c<='Z')
16            {
17                c+=key%26;
18                if(c>'Z') c-=26;
19                if(c<'A') c+=26;
20            }
21            es+=c;
22        }
23        return es;
24    }
25 }
```

CipherAdapter.java

```
1 public class CipherAdapter extends DataOperation
2 {
3     private Caesar cipher;
4
5     public CipherAdapter()
6     {
7         cipher=new Caesar();
8     }
9
10    public String doEncrypt(int key,String ps)
11    {
12        return cipher.doEncrypt(key,ps);
13    }
14 }
```

适配器类（加密适配器类）

配置文件

config.xml

```
1 <?xml version="1.0"?>
2 <config>
3     <className>CipherAdapter</className>
4 </config>
```


XML操作工具类

```
1 import javax.xml.parsers.*;
2 import org.w3c.dom.*;
3 import org.xml.sax.SAXException;
4 import java.io.*;
5 public class XMLUtil
6 {
7     //该方法用于从XML配置文件中提取具体类名，并返回一个实例对象
8     public static Object getBean()
9     {
10         try
11         {
12             //创建文档对象
13             DocumentBuilderFactory dFactory = DocumentBuilderFactory.newInstance();
14             DocumentBuilder builder = dFactory.newDocumentBuilder();
15             Document doc;
16             doc = builder.parse(new File("config.xml"));
17
18             //获取包含类名的文本节点
19             NodeList nl = doc.getElementsByTagName("className");
20             Node classNode=nl.item(0).getFirstChild();
21             String cName=classNode.getNodeValue();
22
23             //通过类名生成实例对象并将其返回
24             Class c=Class.forName(cName);
25             Object obj=c.newInstance();
26             return obj;
27         }
28         catch(Exception e)
29         {
30             e.printStackTrace();
31             return null;
32         }
33     }
34 }
```

Client.java

客户端测试类

```
1 public class Client
2 {
3     public static void main(String args[])
4     {
5         DataOperation dao=(DataOperation)XMLUtil.getBean();
6         dao.setPassword("sunnyLiu");
7         String ps=dao.getPassword();
8         String es=dao.doEncrypt(6,ps);
9         System.out.println("明文为: " + ps);
10        System.out.println("密文为: " + es);
11    }
12 }
```

新的加密算法类

NewCipher.java

```
1 public final class NewCipher
2 {
3     public String doEncrypt(int key,String ps)
4     {
5         String es="";
6         for(int i=0;i<ps.length();i++)
7         {
8             String c=String.valueOf(ps.charAt(i)%key);
9             es+=c;
10        }
11        return es;
12    }
13 }
```

NewCipherAdapter.java

```
1 public class NewCipherAdapter extends DataOperation
2 {
3     private NewCipher cipher;
4
5     public NewCipherAdapter()
6     {
7         cipher=new NewCipher();
8     }
9
10    public String doEncrypt(int key,String ps)
11    {
12        return cipher.doEncrypt(key,ps);
13    }
14 }
```

新的加密算法适配器类

将XML配置文件中<className>节点内容设置为NewCipherAdapter



适配器模式效果与应用

■ 适配器模式优点：

- 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，无须修改原有结构
- 增加了类的透明性和复用性，提高了适配者的复用性，同一个适配器类可以在多个不同的系统中复用
- 灵活性和扩展性非常好
- 类适配器模式：置换一些适配者的方法很方便
- 对象适配器模式：可以把多个不同的适配者适配到同一个目标，还可以适配一个适配者的子类



适配器模式效果与应用

■ 适配器模式缺点：

□ 类适配器模式：

- (1) 一次最多只能适配一个适配者类，不能同时适配多个适配者
- (2) 适配者类不能为最终类
- (3) 目标抽象类只能为接口，不能为类

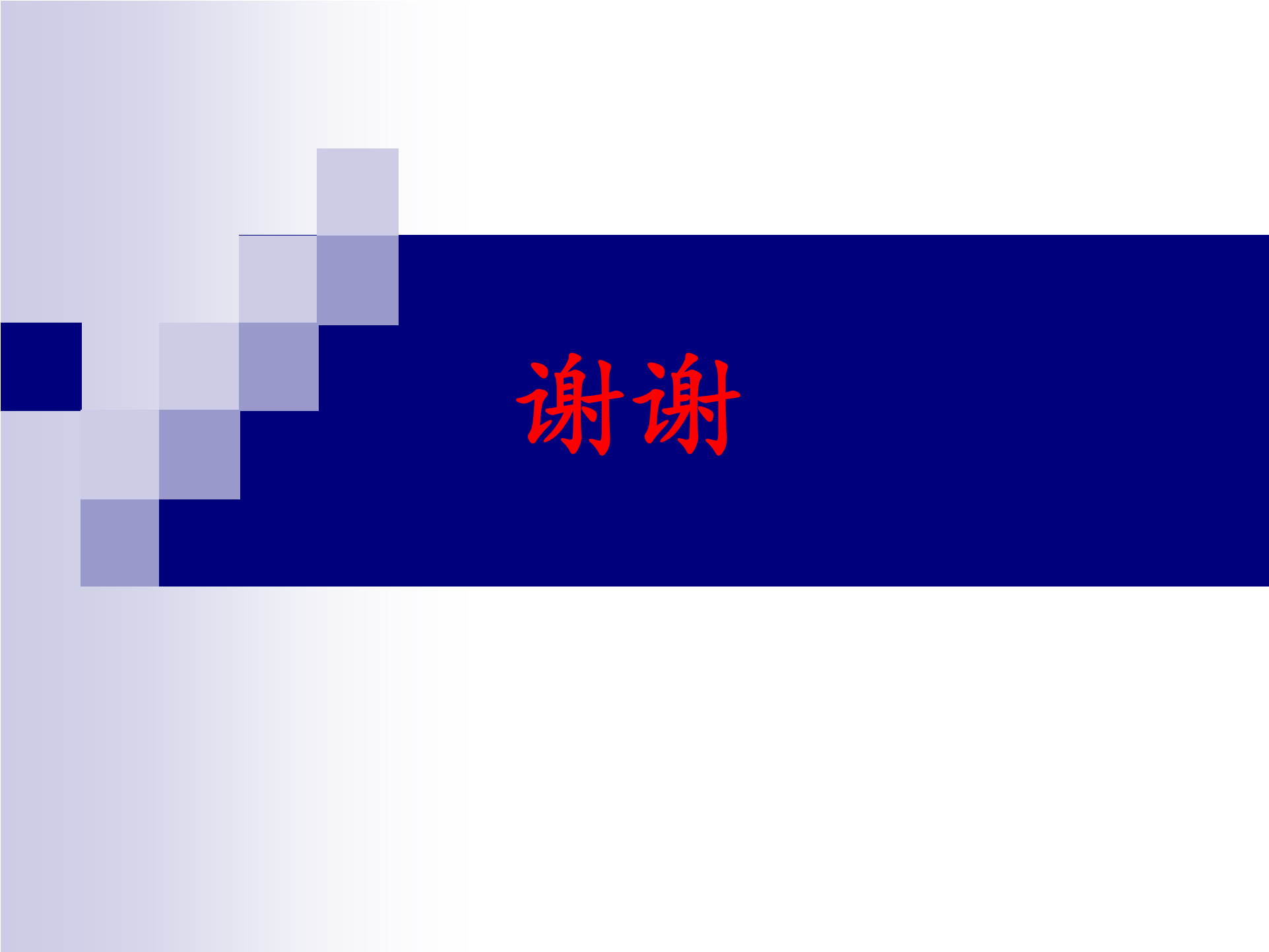
□ 对象适配器模式：在适配器中置换适配者类的某些方法比较麻烦



适配器模式效果与应用

■ 在以下情况下可以使用适配器模式：

- 系统需要使用一些现有的类，而这些类的接口不符合系统的需要，甚至没有这些类的源代码
- 创建一个可以重复使用的类，用于和一些彼此之间没有太大关联的类，包括一些可能在将来引进的类一起工作

The image features a large, solid dark blue horizontal bar that spans most of the width of the slide. To the left of this bar, there is a decorative graphic consisting of a series of squares of varying shades of blue and purple, arranged in a staircase pattern that ascends from the bottom left towards the top right. The squares are semi-transparent, creating a layered effect. The Chinese characters '谢谢' (Thank you) are written in a bold, red, serif font, centered within the dark blue bar.

谢谢