1 java基础 23

1.1 关键字和保留字

关键字和保留字

51 个关键字

基本数据类型、类型的大致范围、越界问题、强转问题

boolean , char , byte , float , int , long , short , double , void , enum

程序控制流

do , while , for , if , else , break , continue , return , switch , case , default

类及接口

class , interface , private , protected , public , implements , extends

修饰词

static , abstract , final , transient , synchronized , volatile , strictfp , native

try , catch , finally , throw , throws , assert , instanceof **异常、断言、类型判断**

文件组织

import , package

new, super, this

3 个直接量 true , false , null

2 个保留字 goto , const

1.2 浅拷贝、深拷贝、引用

- **基本数据类型**（如 `int` `long long` `boolean` ）都是**深拷贝**：复制内容
- **类**的拷贝默认是**浅拷贝**（拷贝引用），需要深拷贝可以使用 `clone()` 方法、序列化机制或手动创建新的对象。

```
class Demo{
    public static void main(String[] args){
        int x = 4;
        show(x);
        System.out.println("x="+x);
    }
    public static void show(int x){
        x = 5;
    }
}
```

【输出】5

```java
public static void main(String[] args) {
    Student s1 = new Student("Test", 5);
    change(s1);
    System.out.println(s1.age);
}
public static void change(Student student){
    student.age = 10;
}
```

【输出】10

1.3 构造与垃圾回收

构造函数

- 特点：与类同名，无返回值（也不是 void）、多数情况要重载
- **new** 的作用：**分配空间；调用构造；返回引用**

不带参数构造函数/默认构造函数

- 如果类的定义者**没有显式的定义任何构造方法**，系统将自动提供一个**默认的构造方法（无参无方法）**；如果定义了构造函数，则不会创建默认构造方法。
- 无参构造函数创建对象时，成员变量的值被赋予了数据类型的**隐含初值**。

带参数构造函数

```java
public class Teacher6 {
        private String name;    // 教员姓名
        private int age;                // 年龄
        private String education;       // 学历
        private String position;        // 职位
        // 带参数的构造方法
        public Teacher6(String pName,int pAge,String pEducation,String
pPosition) {
        name = pName;
        age = pAge;     // 可以增加对age等属性的存取限制条件
        education = pEducation;
        position = pPosition;
    }
    public String introduction() {
        return "大家好！我是" + name + "，我今年" + age + "岁，学历" +
education + "，目前职位是"+position;
    }
}

public class Teacher6Test {
```

```
        public static void main(String[] args) {
        Teacher6 teacher = new Teacher6("Mary", 23, "本科", "咨询师");
        System.out.println(teacher.introduction());
        }
}
```

垃圾内存自动回收机制

- 垃圾自动回收机制（Garbage Collection）：Java 虚拟机后台线程负责
- System.gc() 和 Runtime.gc()
- 判断存储单元是否为垃圾的依据：**引用计数为 0**

1.4 匿名对象

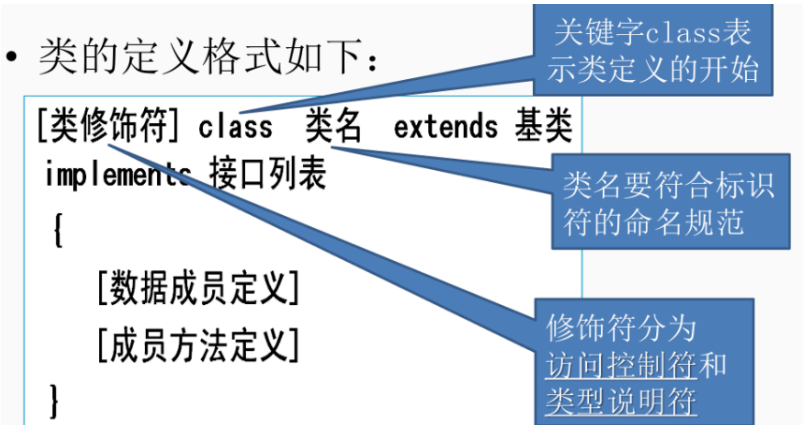
- 匿名对象：无管理者（无栈内存引用指向它）
- 使用场景：**只需要进行一次方法调用 / 作为参数**传递给函数

```
Student stu = new Student();//stu是对象，名字是stu
new Student();//这个也是一个对象，但是没有名字，称为匿名对象
new Student().show(); //匿名对象方法调用
```

1.5 类的定义

类的定义

- 类的定义格式



- 访问控制符：`public` 或默认（即没有访问控制符）
  `public` 类一般含有 `main` 方法
- 类型说明符：`final` 和 `abstract`

成员变量定义

- 定义格式：`[修饰符] 变量的数据类型 变量名[=初始值]`
- 常用的修饰符：`this`、`static`、`public`、`private`、`protected`、默认

成员方法的定义

- 定义格式

```
[修饰符] 返回值类型 方法名([形参说明])[thorws 例外名1，例外名2 ... ]{
        局部变量声明;
        执行语句组;
}
```

- 常用的修饰符：`public`、`private`、`protected`、`static`、`final`

成员变量vs局部变量

- 初始化不同：**自动初始化只用于成员变量**；方法体中的局部变量不能被自动初始化，必须赋值后才能使用。
- 定义的位置不同：定义在类中的变量是成员变量；定义在方法中或者{}语句里面的变量是局部变量。
- 在内存中的位置不同：**成员变量**存储在**堆**内存的对象中；**局部变量**存储在**栈**内存的方法中。
- 声明周期不同：成员变量**随着对象**的出现而出现在堆中，随着对象的消失而从堆中消失；局部变量**随着方法**的运行而出现在栈中，随着方法的弹栈而消失。

方法的重载

1. 方法的重载是指一个类中可以定义有**相同的名字**，但**参数不同**的多个方法，调用时会根据不同的参数表选择对应的方法。
2. 重载方法必须满足以下条件：
   - 方法名相同。
   - 方法的**参数类型、个数、顺序**至少有一项不相同。
   - 方法的**返回类型可以**不相同。
   - 方法的**修饰符可以**不相同。
3. 调用重载方法时，Java使用参数的类型和数量决定实际调用重载方法的哪个版本。

1.6 toString()方法

- 在java中，所有对象都有默认的`toString()`这个方法
- 创建类时没有定义`toString()`方法，输出对象时会输出对象的哈希码值（**对象的内存地址**）
- 它通常只是为了方便输出，比如`System.out.println(xx)`，（xx是对象），括号里面的”xx”如果不是String类型的话，就自动调用xx的`toString()`方法
- `toString()`的定义格式

```
public String toString(){
        return 字符串;              // 方法体
}
```

1.7 静态属性和静态方法

静态成员变量：类加载时就被装载和分配，不依赖于对象而存在 => 通过 类名.变量名 访问
静态成员方法：可以通过 **类名.函数名 调用，只能访问类的静态成员**
静态代码块：仅能定义在类中，一般用于初始化类的静态变量

语句执行顺序

第一次 new 时：**（1）初始化有显式初始化的静态成员变量；（2）顺序执行静态代码块；**（3）初始化有显式初始化的非静态成员变量；（4）顺序执行非静态代码块；（5）调用构造。
之后再 new 时：（1）初始化有显式初始化的非静态成员变量；（2）顺序执行非静态代码块；（3）调用构造。

- 用 `static` 修饰
- 不创建具体对象也存在，此时可以通过 `类名.类变量` 或 `类名.类方法`
- 静态方法与非静态方法的区别
  - 静态方法是在类中使用staitc修饰的方法，在**类定义**的时候已经被**装载和分配**(早加载)。而非静态方法是不加static关键字的方法，在类定义时没有占用内存，只有在类被**实例化成对象时**，对象调用该方法才被**分配内存**（晚加载）。
  - **静态方法**中**只能直接调用静态成员或者方法**，不能直接调用非静态方法或者非静态成员（非静态方法要被实例化才能被静态方法调用），而非静态方法既可以调用静态成员或者方法又可以调用其他的非静态成员或者方法。
- 静态代码块
  - 静态代码块只能定义在类里面，它独立于任何方法，**不能定义在方法里面。**
  - 静态代码块里面声明的变量都是局部变量，只在**本块内有效。**
  - 静态代码块会在**类被加载时自动执行**，而无论加载者是JVM还是其他的类。
  - 一个类中允许定义多个静态代码块，**执行的顺序根据定义的顺序进行。**
  - 静态代码块只能访问类的**静态成员**，而不允许访问实例成员。
- 静态代码块与非静态代码块的不同点：
  - 静态代码块在非静态代码块之前执行：静态代码块—>非静态代码块—>构造方法
  - 静态代码块**只在第一次new执行一次**，之后不再执行，而非静态代码块在每new一次就执行一次

静态变量

```java
package com.dal.staticEx;

class Person {
    String name;//instance variable
    String sex;//instance variable
    int age;  //instance variable
    private static int count;  //类变量class variable 在全局分配内存
    public static int getCount() {  //类方法 class method
        return count;
    }
    public Person(String n, String s, int a) {//constructor
        name = n;
        sex = s;
        age = a;
        count++;
    }
    public String toString() {//instance method
        String s = "姓名：" + name + "," + "性别：" + sex + "," + "年龄：" + age;
        return s;
    }
}
```

```java
package com.dal.staticEx;

public class TestPerson3 {
    public static void main(String[] args) {
        Person p1 = new Person("张三", "男", 20);
        System.out.print("count=" + p1.getCount() + "\t");//1
        System.out.print("count=" + Person.getCount() + "\n");//1
        Person p2 = new Person("Tom", "M", 50);
        System.out.print("count=" + p2.getCount() + "\t");//2
        System.out.print("count=" + Person.getCount() + "\n");//2
        Person p3 = new Person("Mary", "F", 10);
        System.out.print("count=" + p3.getCount() + "\n");//3
        System.out.println("通过类名和不同对象名访问静态变量count:");
        System.out.print("count=" + Person.getCount() + "\n");//3
        System.out.print("count=" + p1.getCount() + "\t");//3
        System.out.print("count=" + p2.getCount() + "\t");//3
        System.out.print("count=" + p3.getCount() + "\n");//3
    }
}
```

例：静态方法

```java
class Test{
        public int sum(int a, int b){     // 非静态方法
                return a+b;
        }
        public static void main(String[] args){
                int result = (sum1, 2);     // 静态方法调用非静态方法，报错
                System.out.println("result="+result);
```

```
        }
}
```

例：静态代码块

```
public class PuTong{
        public PuTong(){
                System.out.println("默认构造方法！⟶");
        }
        // 非静态代码块
        {
                System.out.println("非静态代码块！⟶");
        }
        // 静态代码块
        static{
                System.out.println("静态代码块！⟶");
        }
        // 静态成员方法
        public static void test(){
                System.out.println("普通方法中的代码块！⟶");
        }
        public static void main(String[] args){
                PuTong c1 = new PuTong();
                c1.test();
                PuTong c2 = new PuTong();
                c2.test();
        }
}

/*
【输出】
静态代码块！⟶
非静态代码块！⟶
默认构造方法！⟶
普通方法中的代码块！⟶
非静态代码块！⟶
默认构造方法！⟶
普通方法中的代码块！⟶
*/
```

1.8 equals 和 ==

- 比较**对象**的equals和==是等价的，判断**是不是引用的同一个对象。**
- **String**的**equals**只看**字符串内容**是否相等，而==还得看是不是**同一个对象。**（覆盖了Object类的equals()方法，java.io.File、java.util.Date、包装类（如java.lang.Integer和

java.lang.Double类等）同理）

```
String str1=new String("Hello");
String str2=new String("Hello");

System.out.println(str1==str2); //打印false
System.out.println(str1.equals(str2)); //打印true
```

```
Integer a = 1;
Integer b = 1;
System.out.println(a == b); // 输出 true
```

这行输出 `true` 是因为 `Integer` 有一个缓存机制，对于 `-128` 到 `127` 之间的整数， `Integer` 会缓存这些值的实例。当你使用 `Integer a = 1;` 和 `Integer b = 1;` 这样的方式创建对象时， `a` 和 `b` 实际上指向了缓存中的同一个 `Integer` 实例。因此， `a == b` 比较的是同一个实例的引用，结果为 `true` 。

1. **Byte**：由于Byte的值范围在-128到127之间，所以所有的Byte值都被缓存。
2. **Short**和**Integer**：这两个类的缓存机制类似，都有默认的缓存范围，通常是-128到127。但是这个范围是可以调整的，通过JVM参数可以设置缓存的最大值。
3. **Long**：Long类型也有缓存机制，但是默认的缓存范围比较小，通常是-128到127。同样，这个范围也可以通过JVM参数进行调整。
4. **Character**：Character类型缓存了ASCII字符，即0到127的字符。
5. **Boolean**：Boolean类型比较特殊，它只有两个值true和false，这两个值都是缓存好的。
6. **Float**和**Double**：这两个类型没有缓存机制，因为浮点数的取值范围非常大，缓存所有的值是不现实的

2 封装 4

2.1 封装的含义

1. 一层含义是**把对象的属性和行为看成为一个密不可分的整体**，将这两者封装在一个不可分割的独立单位（即对象）中。
2. 另一层含义指**信息隐藏**，把不需要让外界知道的信息隐藏起来，有些对象的属性及行为允许外界用户知道或使用，但不允许更改，而另一些属性和行为则不允许外界知晓或只允许使用对象的功能，而尽可能隐藏对象的功能实现细节。

2.2 信息隐藏的必要性

成员变量封装加上 `private`，对外提供公开的用于设置对象属性的 `public` 方法，并在方法中加上逻辑判断，过滤掉非法数据，从而：

- **隐藏了类的具体实现**

- **操作简单**
- **提高对象数据的安全性**
- **减少了冗余代码**，数据校验等写在方法里，可以复用

2.3 访问控制修饰符

访问控制分四种类别：

- 公开 `public` 对外公开。
- 受保护 `protected` 向**子类**以及**同一个包中的类**公开。
- 默认 向**同一个包中的类**公开。
- 私有 `private` 只有类本身可以访问，不对外公开

修饰符	同一个类	同一个包	子类	整体
`private`	yeah			
`default`	yeah	yeah		
`protected`	yeah	yeah	yeah	
`public`	yeah	yeah	yeah	yeah

2.3.1 `protected`

1 包内可见

2 子类可见（子类和父类在同一个包：通过自己访问、通过父类访问。在不同包：仅可通过自己访问。）

若子类与父类**不在同一包**中，那么在子类中

- **子类实例**可以访问其从父类继承而来的 `protected` 方法
- 不能访问**父类实例**的 `protected` 方法
- 不能通过**另一个子类引用**访问共同基类的 `protected` 方法。

```
// 父类
package com.protectedaccess.parentpackage;

public class Parent{
        protected String protect = "protect field";
        protected void getMessages(){
                System.out.println("i am parent");
        }
}
```

```
// 不同包下，可以访问其从父类继承而来的`protected`方法
package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;
public class son1 extends Parent {
    public static void main(String[] args) {
        Son1 son1 = new Son1();
        son1.getMessage();  // 输出：i am parent
    }
    private void message() {
        getMessage();  // 如果子类重写了该方法，则输出重写方法中的内容
        super.getMessage();  // 输出父类该方法中的内容
    }
}
```

```
// 不同包下，不能访问**父类实例**的`protected`方法
package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;
public class son1 extends Parent {
        public static void main(String[] args) {
                Parent parent1 = new Parent();
                // parent1.getMessage(); // 错误
                Parent parent2 = new Parent();
                // parent2.getMessage(); // 错误
        }
}
```

```
// 不同包下，不能通过**另一个子类引用**访问共同基类的`protected`方法。
package com.protectedaccess.parentpackage.sonpackage2;
import com.protectedaccess.parentpackage.Parent;
public class son2 extends Parent {

}
```

```
package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;
import com.protectedaccess.parentpackage.sonpackage2.Son2;
public class son1 extends Parent {
    public static void main(String[] args) {
        Son2 son2 = new Son2();
        // son2.getMessage();  // 错误
    }
}
```

3 protected 的 static 成员对所有子类可见。

对于`protected`修饰的静态变量，无论是否同一个包，在子类中均可直接访问；在不同包的非子类中则不可访问。

```
// 父类
package com.protectedaccess.parentpackage;

public class Parent{
        protected String protect = "protect field";
        protected static void getMessages(){
                System.out.println("i am parent");
        }
}
```

```
// 无论是否同一个包，在子类中均可直接访问
package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;

public class son3 extends Parent {
    public static void main(String[] args) {
        Parent.getMessage();    // 输出：i am parent
    }
}
```

```
// 在不同包的非子类中则不可访问
package com.protectedaccess.parentpackage.sonpackage1;
import com.protectedaccess.parentpackage.Parent;

public class son4 {
    public static void main(String[] args) {
        // Parent.getMessage();    // 错误
    }
}
```

2.4 例

2. 以下关于构造方法的说明不正确的是：_____
A 构造方法名必须和类名相同
B 构造方法不返回任何值，也没有返回类型
C 每一个类只能有一个构造方法
D 构造方法在创建对象时调用，其他地方不能显式地直接调用
正确答案: C

8. 下列关于权限的说法中错误的是：_____
A Java中一共有四种访问权限控制，其权限的大小顺序：public > protected > default > private
B 普通方法可以使用四种访问权限，但是抽象方法不能使用private来修饰
C 构造器可以使用四种访问权限
D protected所修饰的成员，对于同包子类可访问，但是外包子类不可访问
正确答案: D

3 继承 5

- **定义**: 继承是允许创建新的类（子类）来继承现有类（父类）的属性和行为的能力。
- **目的**:
  - **代码复用**: 子类可以继承父类的所有属性和方法，无需重复编写代码，提高代码的可重用性。
  - **扩展功能**: 子类可以添加新的属性和方法，或重写父类的方法，扩展父类的功能。
  - **层次结构**: 继承可以创建类层次结构，清晰地表达类之间的关系，使代码更易于理解和维护。
- **类型**:

- **单继承**: 一个子类只有一个父类。
- **多继承**: 一个子类可以有多个父类（Java 不支持多继承）。
- **接口继承**: 子接口可以继承父接口的方法，并添加新的方法。

继承（Inheritance)

继承虽好，但也破坏了封装性。子类(派生类)拥有父类(基类/超类)的**所有属性和方法**
(不过私有的属性和方法不能直接访问)

构造方法不能继承

子类构造方法必须最先调用父类构造。

new 出子类对象时，构造方法按照继承链从上往下依次调用。

变量隐藏：子类隐藏父类的同名变量。

继承：super vs this

共性

（1）出现在实例方法和构造方法中
（2）不能出现在静态方法中
（3）大部分情况下是可以省略的
（4）this(), super() 只能出现在构造方法的第一行（若没有 自动补上无参构造方法）

super

（1）当在子类对象中，子类想访问父类的东西，可以使用“super.”的方式访问
（2）当子类构造方法的第一行执行super()无参数方法，那么父类中一定要有无参数构造方法（当一个类的构造方法是显式的有参数的，会替代原本的无参构造方法，这个时候如果子类执行super()无参方法就会报错，一个好的习惯是要补上无参的构造方法）

继承：方法重写 / 覆盖（Override）

条件

（1）子类中方法的返回值必须与父类的相同、或是其子类。
（2）**方法名、形参列表**完全相同。
（3）**访问权限**不能更低，只能相同或更高。
（4）抛出异常的范围不能更大。

注意事项

（1）**private 方法**不能继承，所以不能覆盖，只有隐藏。
（2）**构造方法**不能继承，所以不能覆盖。
（3）**静态方法**不存在覆盖，只有隐藏。
（4）**final 修饰**的是最终方法，不能覆盖。

4 多态 6

- **定义**: 多态是指同一个行为具有多个不同表现形式或形态的能力。
- **目的**:
  - **灵活性**: 允许以统一的方式处理不同类型的对象，提高代码的灵活性和可扩展性。
  - **抽象性**: 可以隐藏对象的实际类型，只关注其共同的行为，提高代码的抽象性。
- **实现方式**:
  - **方法重载**: 在同一个类中，可以有多个同名但参数类型或数量不同的方法。
  - **方法重写**: 子类可以重写父类的方法，提供不同的实现。
  - **接口**: 接口定义了一组方法，不同的类可以实现同一个接口，并提供不同的实现。

4.1 多态

- 多态：**用相同的名称来表示不同的含义**
- **静多态**：在**编译**时决定调用哪个方法；**方法重载、方法隐藏**

- **方法重载(Overloading)**
  - 方法名相同，**参数个数**、**参数类型**及**参数顺序**至少有一个不同
  - 返回值类型与访问权限修饰符可以相同也可以不同

```
public void print(String name, int age) {
    System.out.println("Name: " + name + ", Age: " + age);
}
public void print(int age) {
    System.out.println("Age: " + age);
}
```

```
private void show(String msg) {
    System.out.println("Private method: " + msg);
}
public void show(String msg) {
    System.out.println("Public method: " + msg);
}
```

- **方法隐藏**：子类定义了一个与父类**同名同参数列表**的**静态/私有**方法（相当于一个新方法）

```
class Parent {  1 个继承者
    public static void priFinalMethod(){  2 个用法
        System.out.println("PersonNNNNNNNNNNNNNNN");
    }
    public static void main(String[] args) {
        Parent p = new Parent();
        Child c = new Child();
        Parent p2 = new Child();
        p.priFinalMethod();
        c.priFinalMethod();
        p2.priFinalMethod();
    }
}

class Child extends Parent {
    public static void priFinalMethod(){  1 个用法
        System.out.println("StudentTTTTTTTTTTTTTTTTTTT");
    }
}
```

```
Parent
PersonNNNNNNNNNNNNNNN
StudentTTTTTTTTTTTTTTTTTTT
PersonNNNNNNNNNNNNNNN
```

- **动多态**：在**运行**时才能确定调用哪个方法；**方法覆盖**
  - 3个条件：**继承、覆盖、向上转型**（必须由父类的引用指向派生类的实例，并且通过父类的引用调用被覆盖的方法）

- **方法覆盖(Override)**
  - 方法名、参数个数、参数类型及参数顺序必须一致
  - **异常抛出范围**：子类 ≤ 父类
  - **访问权限**：子类 ≥ 父类
  - **私有方法、静态方法不能被覆盖**，如果在子类出现了同签名的方法，那是方法隐藏；

- **多态中成员变量编译运行看左边，多态中成员方法编译看左边，运行看右边**

```
// **多态中成员变量编译运行看左边**
class Parent {
    int value = 10;
}

class Child extends Parent {
    int value = 20;
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        System.out.println(p.value); // 输出 10，因为编译和运行时都看左边（Parent类）
    }
}
```

```
// **多态中成员方法编译看左边，运行看右边**
class Parent {
    void display() {
        System.out.println("Parent display");
    }
}

class Child extends Parent {
    @Override
    void display() {
        System.out.println("Child display");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent p = new Child();
        p.display(); // 编译时看左边（Parent类），运行时看右边（Child类），输出 "Child display"
```

```
    }
}
```

- 抽象类
  - `abstract` 语义为“尚未实现”
  - 如果一个类继承自某个抽象父类，而没有具体实现抽象父类中的抽象方法，则必须定义为抽象类
  - 抽象类引用：虽然**不能实例化抽象类**，但可以**创建它的引用**。因为Java支持多态性，允许通过父类引用来引用子类的对象。
  - 如果一个类里有抽象的方法，则这个类就必须声明成抽象的。**但一个抽象类中却可以没有抽象方法。**
- 抽象方法
  - 无函数体
  - 必须在抽象类中
  - 必须在子类中实现，除非子类也是抽象的
  - 不能被**private、final或static**修饰。

4.2 接口

4.2.1 定义

- 接口：不相关类的功能继承。
  - 只包含**常量**（所有变量默认`public static final`）和**方法**（默认`public abstract`）的定义，没有方法的实现。（但一般不包含变量）
  - **没有构造方法**
  - 一个类可以实现多个接口；如果类没有实现接口的全部方法。需要被定义成`abstract`类
  - 接口的方法体还可以由其他语言写，此时接口方法需要用`native`修饰
  - 接口可以继承，而且可以多重继承
    - 同一个函数只能实现一次
    - 不同接口的同名变量相互隐藏
    - 接口变量和类中成员同名时，存在作用域问题
- 关键词`interface` `implements`

```
[public] [interface] 接口名称 [extends 父接口名列表]{
        // 静态常量
        [public][static][final]数据类型 变量名=常量名;
        //抽象方法
        [public][abstract][native]返回值类型 方法名（参数列表）;
}

[修饰符] class类名 [extends父类名] [implements接口A,接口B,…]{
        类的成员变量和成员方法;
```

```
        为接口A中的所有方法编写方法体，实现接口A;
        为接口B中的所有方法编写方法体，实现接口B;
}
```

```
public interface Flyer{
        public void takeOff();
        public void land();
        public void fly();
}

public class Bird extends Animal implements Flyer {
        public void takeOff() { /* take- off implementation */ }
        public void land() { /* landing implementation */ }
        public void fly() { /* fly implementation */ }
        public void buildNest() { /* nest building behavior */ }
        public void layEggs() { /* egg laying behavior */ }
        public void eat() { /* override eating behavior */ }
}
```

```
// 不同接口的同名变量相互隐藏
interface Animal {
    int legs = 0; // 假设动物默认没有腿
}

interface 昆虫 {
    int legs = 6; // 昆虫有6条腿
}

class Spider implements Animal,昆虫 {
    public void printLegs() {
        System.out.println("Animal legs: " + Animal.legs); // 调用
Animal接口中的legs变量
        System.out.println("昆虫 legs: " +昆虫.legs); // 调用昆虫接口中的
legs变量
    }
}

public class Main {
    public static void main(String[] args) {
        Spider spider = new Spider();
        spider.printLegs();
    }
}
```

```
// 接口变量和类中成员同名时，存在作用域问题
interface Animal {
```

```
    String name = "Animal";
}

class Dog implements Animal {
    String name = "Dog"; // Dog类中定义了与接口同名的变量

    public void printName() {
        System.out.println(name); // 这将打印"Dog"
        System.out.println(Animal.name); // 这将打印"Animal"
    }
}
```

4.2.2 使用接口

- 接口用作类型
  - 声明格式：`接口 变量名` (又称为引用)
  - 接口做参数：如果一个方法的参数是接口类型，就可以将任何实现该接口的类的实例的引用传递给接口参数，那么接口参数就可以回调类实现的接口方法。
- 接口回调
  - 把实现某一接口的类创建的**对象引用**赋给该接口声明的**接口变量**
  - 该接口变量就可以**调用被类实现的接口中的方法。**
  - 即：

    `接口变量 = 实现该接口的类所创建的对象；`

    `接口变量.接口方法([参数列表])`;

```
interface Runner {
        //接口 1
        public void run();
}

interface Swimmer {
        //接口 2
        public void swim();
}

abstract class Animal {
        public abstract void eat();
}

class Person extends Animal implements Runner,Swimmer { //继承类，实现接口
        public void run() {
                System.out.println("我是飞毛腿,跑步速度极快!");
        }
        public void swim(){
                System.out.println("我游泳技术很好,会蛙泳、自由泳、仰泳、蝶
```

```
泳 ... ");
        }
        public void eat(){
                System.out.println("我牙好胃好,吃啥都香!");
        }
}

public class InterfaceTest{
        public void m1(Runner r) { r.run(); } //接口作参数
        public void m2(Swimmer s) {s.swim();} //接口作参数
        public void m3(Animal a) {a.eat();} //抽象类引用
        public static void main(String args[]){
                InterfaceTest t = new InterfaceTest();
                Person p = new Person();
                t.m1(p); //接口回调
                t.m2(p); //接口回调
                t.m3(p); //接口回调
        }
}
```

4.2.4 抽象类与接口

- 区别
  - 接口中的成员**变量和方法**只能是 `public` 类型的，而抽象类中的成员变量和方法可以处于各种访问级别。
  - 接口中的**成员变量**只能是 `public`、`static` 和 `final` 类型的，而在抽象类中可以定义各种类型的实例变量和静态变量。
  - 接口中没有**构造方法**，抽象类中有构造方法。接口中所有方法都是抽象方法，抽象类中可以有，也可以没有抽象方法。抽象类比接口包含了更多的实现细节。
  - 抽象类是某一类事物的一种抽象，而接口不是类，它**只定义了某些行为**；
    例如，“生物”类虽然抽象，但有“狗”类的雏形，接口中的run方法可以由狗类实现，也可以由汽车实现。
  - 在语义上，接口表示更高层次的抽象，声明系统对外提供的服务。而抽象类则是各种具体类型的抽象。

4.2.5 Native关键字

- Native用来声明一个方法是由机器相关的语言（如C/C++语言）实现的。通常，native方法用于一些比较消耗资源的方法，该方法用c或其他语言编写，可以提高速度。
- native 定义符说明该方法是一个使用本地其他语言编写的非java类库的方法，它是调用的本地（也就是当前操作系统的方法或动态连接库）。最常见的就是c/c++封装的DLL里面的方法，这是java的 JNI技术。它在类中的声明和抽象方法一样没有方法体。

4.3 upcasting 和 downcasting

4.3.1 向上转型 upcasting

- 向上转型：当有**子类对象赋值给一个父类引用**时，便是向上转型，多态本身就是向上转型的过程。

- 使用格式：
  父类类型 变量名 = new 子类类型();
  如：Person p = new Student();

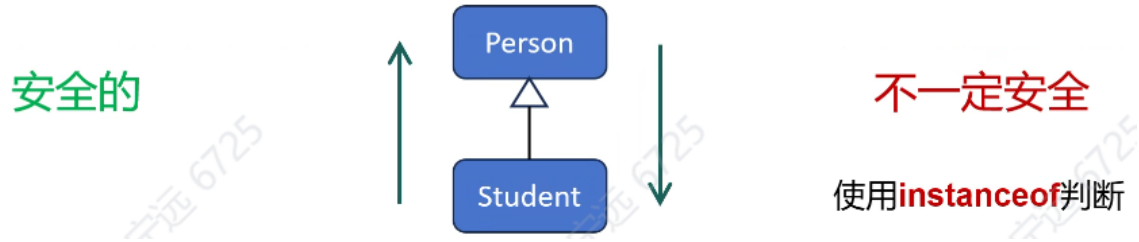
- 上转型对象的使用(父类有的就能访问，没有的不能访问，且儿子的优先级更高)
  - 上转型对象可以**访问子类继承或隐藏的成员变量**，也可以**调用子类继承的方法或子类重写的实例方法。**
  - 如果子类重写了父类的某个实例方法后，当用上转型对象调用这个实例方法时一定是**调用了子类重写的实例方法。**
  - 上转型对象**不能操作子类新增**的成员变量；**不能调用子类新增**的方法。

4.3.2 向下转型 downcasting

- 向下转型(映射)：一个**已经向上转型的子类对象**可以使用强制类型转换的格式，**将父类引用转为子类引用**，这个过程是向下转型。
- 使用格式：
  `子类类型 变量名 = (子类类型) 父类类型的变量;`
  如：Person p = new Student();
  Student stu = (Student) p
- 如果是**直接创建父类对象，是无法向下转型的**，能过编译，但运行时会产生异常
  如：Person p = new Peron();
  Student stu = (Student) p

instanceof 操作符

- `instanceof` 操作符用于判断一个引用类型所**引用的对象是否是一个类的实例。**
  `instanceof` 运算符是Java独有的双目运算符
- `instanceof` 操作符左边的操作元是一个引用类型的对象（可以是null），右边的操作元是一个类名或接口名。
- 形式如下：`obj instanceof ClassName` 或者 `obj instanceof InterfaceName`
- `a instanceof X`，当 X 是 **A类/A类的直接或间接父类/A类实现的接口**时，表达式的值为true



5 Object类、最终类、内部类、匿名类 10

5.1 Object类

- 基于多态的特性，该类可以用来代表任何一个类，因此允许**把任何类型的对象赋给Object类型的变量**，也可以作为方法的参数、方法的返回值

Object类提供的几个关键方法

方法摘要	
protected Object	clone() 创建并返回此对象的一个副本。
boolean	equals(Object obj) 指示其他某个对象是否与此对象“相等”。
protected void	finalize() 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
Class<?>	getClass() 返回此 Object 的运行时类。
int	hashCode() 返回该对象的哈希码值。
void	notify() 唤醒在此对象监视器上等待的单个线程。
void	notifyAll() 唤醒在此对象监视器上等待的所有线程。
String	toString() 返回该对象的字符串表示。
void	wait() 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。
void	wait(long timeout) 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。
void	wait(long timeout, int nanos) 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，

`public final Class<?> getClass(){}`

- 该方法用于获取对象运行时的字节码类型，得到该对象的**运行时的真实类型。**
- 通常用于判断**两个引用中实际存储对象类型**是否一致。

```
public class MyClass {
    // 类的成员变量和方法
}

public class AnotherClass {
    // 另一个类的成员变量和方法
}

public class TypeComparisonExample {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();
        AnotherClass obj3 = new AnotherClass();

        // 比较obj1和obj2是否指向相同类型的对象
```

```
        if (obj1.getClass() == obj2.getClass()) {
            System.out.println("obj1和obj2指向相同类型的对象");
        } else {
            System.out.println("obj1和obj2指向不同类型的对象");
        }

        // 比较obj1和obj3是否指向相同类型的对象
        if (obj1.getClass() == obj3.getClass()) {
            System.out.println("obj1和obj3指向相同类型的对象");
        } else {
            System.out.println("obj1和obj3指向不同类型的对象");
        }
    }
}

// obj1和obj2指向相同类型的对象
// obj1和obj3指向不同类型的对象
```

- 最主要应用：该方法属于Java的反射机制，其返回值是Class类型，例如 Class c = obj.getClass();。通过对象c，
  - 获取所有**成员方法**，**每个成员方法都是一个Method对象。** `Method[] methods = cls.getDeclaredMethods();`
  - 获取所有**成员变量**，**每个成员变量都是一个Field对象。** `Field[] fields = cls.getDeclaredFields();`
  - 获取所有**构造函数**，**构造函数则是一个Constructor对象。** `Constructor<?>[] constructors = cls.getDeclaredConstructors();`

```
// 用getClass来获得该对象的类名、所有成员方法、所有成员变量、所有构造函数
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class MyClass {
    private int myField;
    public MyClass() {
        // 默认构造函数
    }
    public MyClass(int value) {
        // 带参数的构造函数
        myField = value;
    }
    public void myMethod() {
        // 成员方法
    }
}
```

```java
public class ReflectionExample {
    public static void main(String[] args) {
        MyClass obj = new MyClass();

        // 获取Class对象
        Class<?> cls = obj.getClass();

        // 获取类名
        String className = cls.getName();
        System.out.println("类名: " + className);

        // 获取所有成员方法
        Method[] methods = cls.getDeclaredMethods();
        System.out.println("成员方法:");
        for (Method method : methods) {
            System.out.println(method.getName());
        }

        // 获取所有成员变量
        Field[] fields = cls.getDeclaredFields();
        System.out.println("成员变量:");
        for (Field field : fields) {
            System.out.println(field.getName());
        }

        // 获取所有构造函数
        Constructor<?>[] constructors = cls.getDeclaredConstructors();
        System.out.println("构造函数:");
        for (Constructor<?> constructor : constructors) {
            System.out.println(constructor.getName());
        }
    }
}
```

`public int hashCode(){}`

- 返回该对象的**哈希码值**。哈希值为根据对象的**地址或字符串或数字**使用hash算法计算出来的**int**类型的数值。
- 在Object类中，hashCode的默认实现通常会返回对象的**内存地址的某种形式**（不能完全将哈希值等价于地址），具体的实现依赖JVM。
- 提高具有哈希结构的容器的效率。
- 如果**两个对象相等**（即equals返回true），那么它们的**hashCode值也必须相等**。

`public boolean equals(Object obj)`

- 比较两个对象是否相等。仅当被比较的两个**引用变量指向同一对象**时（即两个对象地址相同，也即hashCode值相同），equals()方法返回true

- 可进行**覆盖**，**比较两个对象的内容是否相同。**

equals()与hashCode

equals为true与hashCode相同的关系?

- 如果两个对象的equals()结果为**true**，那么这两个对象的**hashCode()一定相同**;
- 两个对象的**hashCode()结果相同**，并**不能代表**两个对象的equals()一定为true（Hash散列值有冲突的情况，虽然概率很低，只能够说明这两个对象在一个散列存储结构中）

为什么要重写hashCode和equals?

- equals() 方法用于比较两个对象的内容是否相等。在Java中，默认实现是比较对象的引用，即比较两个对象是否指向内存中的相同位置。但**通常，我们希望比较对象的内容是否相等。**
- 鉴于这种情况，Object类中 equals() 方法的默认实现是没有实用价值的，所以**通常都要重写。**
- 而由于hashCode()与equals()具有**联动关系**（如果两个对象相等，则它们必须有相同的哈希码），所以equals()方法重写时，通常也要将hashCode()进行重写，使得这两个方法始终保持一致性。

重写equals一定要重写hashCode吗?

- 如果仅仅是为了**比较两个对象是否相等**只重写**equals**就可以;
- 如果你使用了**hashSet、hashMap**等容器，为了避免加入重复元素，或者查找元素，就一定要同时重写两个方法。
- 如果**自定义对象作为 Map 的键**，那么必须重写 hashCode 和 equals 。

例

```
package com.pack1;

public class Test {
        public static void main(String[] args) {
                Person person1 = new Person();
                Person person2 = new Person();
                Person person3 = person2;

                System.out.println("person1 的 hashcode: " + person1.hashCode());
                System.out.println("person2 的 hashcode: " + person2.hashCode());
                System.out.println("person3 的 hashcode: " + person3.hashCode());
        }
}

class Person {

}
```

```
person1 的 hashcode: 1163157884
person2 的 hashcode: 1956725890
person3 的 hashcode: 1956725890
```

如果不重写 hashCode() 和 equals() 方法

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

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
 public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("John", 25);
        Person p2 = new Person("John", 25);
        // 输出：false，默认比较的是内存地址
        System.out.println(p1.equals(p2));
        // 输出：false，hashCode 基于内存地址
        System.out.println(p1.hashCode() == p2.hashCode());
    }
}
```

默认情况下，equals() 和 hashCode() 是基于对象的内存地址比较的

重写 equals() 但不重写 hashCode()

```
class Person {
    String name; int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;}
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return age == person.age && name.equals(person.name);
    }}
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("John", 25);
        Person p2 = new Person("John", 25);
        System.out.println(p1.equals(p2)); // 输出：true，内容相同
        System.out.println(p1.hashCode() == p2.hashCode()); // 输出：false，hashCode 不相等
        // 放入 HashSet 中
        HashSet<Person> set = new HashSet<>();
        set.add(p1);
        set.add(p2);
        System.out.println(set.size()); // 输出：2，应该是1（因为 equals 相等），但 hashCode 不同导致重复
    }
}
```

重写 equals() 和 hashCode()

```
class Person {
    String name;int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age; }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return age == person.age && name.equals(person.name);
    }
    @Override
    public int hashCode() {
        return name.hashCode() + age; // 基于对象的属性计算 hashCode
    }}
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("John", 25);
        Person p2 = new Person("John", 25);
        System.out.println(p1.equals(p2)); // 输出：true
        System.out.println(p1.hashCode() == p2.hashCode()); // 输出：true
        // 放入 HashSet 中
        HashSet<Person> set = new HashSet<>();
        set.add(p1);
        set.add(p2);
        System.out.println(set.size()); // 输出：1，说明 p1 和 p2 被认为是相同的对象
    }}
```


`public String toString(){}`

默认的 `toString()` 输出**包名**加**类名**和**堆上的首地址**

`finalize()`方法

不存在对该对象的其他引用时，由对象的垃圾器调用此方法

线程中常用的方法

- `public final void wait()`：多线程中等待功能
- `public final native void notify()`：多线程中唤醒功能
- `public final native void notifyAll()`：多线程中唤醒所有等待线程的功能

例

```java
package com.buaa.edu.cn;
class A {
    public String toString() {
        return "A is for ObjectTest class";
    }
}
class B {
}
public class ObjectTest {
    public static void main(String[] s) {
        A a1 = new A();
        A a2 = new A();
        A a3 = a1;
        B b1 = new B();
        System.out.println("a1.equals(a1) is " + a1.equals(a1));
        System.out.println("a1.equals(a2) is " + a1.equals(a2));
        System.out.println("a1.equals(a3) is " + a1.equals(a3));
        System.out.println(a1.toString());
        System.out.println("a1 is a instance of class "
                + a1.getClass().getName());
        // /默认的toString()输出包名加类名和堆上的首地址
        System.out.println(b1.toString());
        System.out.println("b1 is a instance of class "
                + b1.getClass().getName());
    }
}
```

```
a1.equals(a1) is true
a1.equals(a2) is false
a1.equals(a3) is true
A is for ObjectTest class
a1 is a instance of class com.buaa.edu.cn.A
com.buaa.edu.cn.B@15db9742
b1 is a instance of class com.buaa.edu.cn.B
```

Objects 与 Object 区别

- Objects 【public final class Objects extends Object】是Object 的工具类，位于java.util包。它从jdk1.7开始才出现，被final修饰不能被继承，拥有私有的构造函数。此类包含static实用程序方法，用于**操作对象或在操作前检查某些条件。**
  - **null** 或 null方法；
  - 用于计算**一堆对象的混合哈希代码**；
  - 返回对象的**字符串**（会对null进行处理）；

- 比较两个对象，以及检查索引或子范围值是否超出范围

变量和类型	方法	描述
static int	checkFromIndexSize(int fromIndex, int size, int length)	检查是否在子范围从 fromIndex（包括）到 fromIndex + size（不包括）是范围界限内 0（包括）到 length（不包括）。
static int	checkFromToIndex(int fromIndex, int toIndex, int length)	检查是否在子范围从 fromIndex（包括）到 toIndex（不包括）是范围界限内 0（包括）到 length（不包括）。
static int	checkIndex(int index, int length)	检查 index是否在 0（含）到 length（不包括）范围内。
static <T> int	compare(T a, T b, Comparator<? super T> c)	如果参数相同则返回0，否则返回 c.compare(a, b)。
static boolean	deepEquals(Object a, Object b)	返回 true如果参数是深层相等，彼此 false其他。
static boolean	equals(Object a, Object b)	返回 true如果参数相等，彼此 false其他。
static int	hash(Object... values)	为一系列输入值生成哈希码。
static int	hashCode(Object o)	返回非的哈希码 null参数，0为 null的论点。
static boolean	isNull(Object obj)	返回 true如果提供的参考是 null，否则返回 false。
static boolean	nonNull(Object obj)	返回 true如果提供的参考是非 null否则返回 false。
static <T> T	requireNonNull(T obj)	检查指定的对象引用是否不是 null。
static <T> T	requireNonNull(T obj, String message)	检查指定的对象引用是否为null，如果是，则抛出自定义的NullPointerException。
static <T> T	requireNonNull(T obj, Supplier<String> messageSupplier)	检查指定的对象引用是否为null，如果是，则抛出自定义的NullPointerException。
static <T> T	requireNonNullElse(T obj, T defaultObj)	如果它是非 null，则返回第一个参数，否则返回非 null第二个参数。
static <T> T	requireNonNullElseGet(T obj, Supplier<? extends T> supplier)	如果它是非 null，则返回第一个参数，否则返回非 null值 supplier.get()。
static String	toString(Object o)	返回调用的结果 toString对于非 null参数， "null"为 null的说法。
static String	toString(Object o, String nullDefault)	如果第一个参数不是 null，则返回在第一个参数上调用 toString的结果，否则返回第二个参数。

5.2 最终类、最终方法、常量

5.2.1 最终类、最终方法

- **最终类**：如果一个类**没有必要再派生子类**，通常可以用final关键字修饰，表明它是一个最终类
- **最终方法**：用关键字final修饰的方法称为最终方法。最终方法既不能被覆盖，也不能被重载，它是一个最终方法，其方法的定义永远不能改变
- final类中的方法可以不声明为final方法，但实际上**final类中的方法都是隐式的final方法**
- final修饰的方法不一定要存在于final类中。
- 定义类头时，**abstract和final不能同时使用**
- 访问权限为**private的方法默认为final**的

5.2.2 常量

- Java中的常量使用关键字 `final` 修饰。
- final既可以修饰**简单数据**类型，也可以修饰**复合数据**类型。
  - 简单数据类型：值不能再变
  - 符合数据类型：引用不能再变，值可以改变
- final常量可以在**声明的同时赋初值**，也可以在**构造函数**中

- 常量既可以是**局部常量**，也可以是**类常量和实例常量**。如果是类常量，在**数据类型前加static修饰**（由所有对象共享）。如果是实例常量，就**不加static修饰**。
- 常量名一般大写，多个单词之间用下划线连接。

局部常量、类常量、实例常量

1. 局部常量（Local Constant）： 局部常量是在**方法、构造器或代码块内部**定义的常量。它们只在定义它们的代码块内部有效，一旦代码块执行完毕，局部常量就不再存在。局部常量通常使用 `final` 关键字来声明，表示其值在初始化后不能被改变。

```
public void myMethod() {
    final int LOCAL_CONSTANT = 10; // 局部常量
    // 这里可以使用 LOCAL_CONSTANT
}
// 这里不能使用 LOCAL_CONSTANT，因为它只在 myMethod 方法内部有效
```

2. 类常量（Class Constant）： 类常量是在**类的静态初始化块或静态成员变量**中定义的常量。它们属于类本身，而不是类的实例。类常量也通常使用 `final` 关键字来声明，并且是 `static` 的，这意味着它们是类的所有实例共享的。

```
public class MyClass {
    public static final int CLASS_CONSTANT = 20; // 类常量
    // 这里可以使用 CLASS_CONSTANT
}
```

3. 实例常量（Instance Constant）： 实例常量是在类的**非静态成员变量**中定义的常量。它们属于类的每个实例，每个实例都有自己的实例常量副本。实例常量同样使用 `final` 关键字来声明，表示一旦被初始化，其值就不能改变。

```
public class MyClass {
    public final int INSTANCE_CONSTANT = 30; // 实例常量
    // 这里可以使用 INSTANCE_CONSTANT
}
```

```java
class PersonA {
    String name;
    String sex;
    int age;
    final static double PAI = 3.1415926; // 静态常量，存放在全局数据区
    final double ID; // 常量，表示每一个人的id不同，但一旦赋值又是不能变化的

    public PersonA(String n, String s, int a, int id) {
        name = n;
        sex = s;
        age = a;
        ID = id; // 构造函数中赋值
    }

    public String toString() {
        String s =
    "姓名：" + name + "," + "性别：" + sex + "," + "年龄：" + age;
        return s;
    }
}

public class FinalTest {
    public static void main(String args[]) {
        final double PAI = 3.1415926; // 局部常量

        // final既可以修饰简单数据类型，也可以修饰符和数据类型
        final PersonA p1 = new PersonA("Tom", "M", 23, 001);
        PersonA p2 = new PersonA("Mary", "F", 20, 002);
        System.out.println("final p1:" + p1.toString());

        // p1=p2 //对final对象重新赋值会产生编译错误
        // 以下对final对象中的成员变量，重新赋指是可以的
        p1.name = p2.name;
        p1.sex = p2.sex;
        p1.age = p2.age;
        System.out.println("final p1:" + p1.toString());
    }
}
```

例

```java
package com.buaa.edu;
public class EduBackground {

    String primarySchool;
    String secondarySchool;
    String juniorHSchool;
    String seniorHSchool;
    String university;

    public EduBackground() {

    }
}
```

```java
package com.buaa.edu;
public class Person {
    private String name;
    private int age;
    private String gender;
    private final EduBackground edu = new EduBackground();
    public Person() {
    }
    // final修饰局部变量、修饰成员方法、修饰方法的参数
    // 修饰局部变量时，局部变量的值不能改变
    public void finalLocal() {
        final int i;
        final EduBackground edu = new EduBackground();
        i = 1;
        System.out.println("finalLocal: i = " + i);
    }
    // 修饰方法的参数时(简单数据类型)，参数i不能被修改
    public void finalArgs(final int i) {
        // i = 3;
        System.out.println("finalArgs: i = " + i);
    }
    // 修饰方法的参数时(复合数据类型)，不能指向新的位置
    public void finalArgs(final EduBackground edu) {
        // edu = new EduBackground();
        System.out.println("finalArgs: edu");
    }

    // 修饰成员方法时，成员方法不能被子类重写
    public final void finalMethod() {
        int i = 2;
        System.out.println("finalMethod: i = " + i);
    }

    private final void priFinalMethod() {
        System.out.println("Person:priFinalMethod");
    }

    public static void main(String[] args) {
        Person per = new Person();
        Student stu = new Student();
        Person per1 = stu;

        per.priFinalMethod();
        stu.priFinalMethod();
        per1.priFinalMethod();
    }
}
```

```java
package com.buaa.edu;
public class Student extends Person {
    private final int stuNumber;
    private int score;
    private static final int BAN_JI=20210001;
    public Student() {
        stuNumber=(int)Math.random()*500;
        score=(int)Math.random()*100;
    }
//子类不能重写父类被final修饰的方法
//  public final void finalMethod() {
//          int i = 2;
//          System.out.println("finalMethod: i = " + i);
//  }
    public final void priFinalMethod() {
        System.out.println("Student:priFinalMethod");
    }
}
```

输出

```
PersonNNNNNNNNNNNNNNN
StudentTTTTTTTTTTTTTTTTTTTT
PersonNNNNNNNNNNNNNNN
```

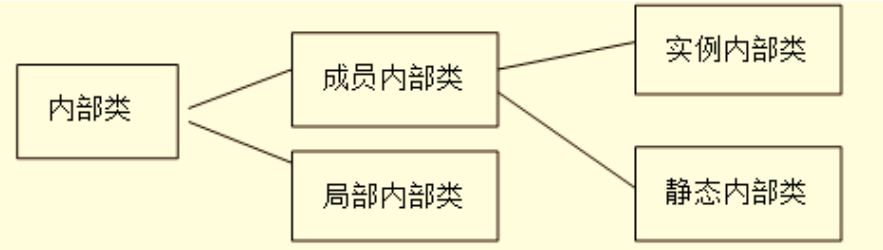
解释

`p.priFinalMethod()` 和 `p2.priFinalMethod()` 调用的是 `Parent` 类的 `priFinalMethod` 方法，因为 `priFinalMethod` 是 `Parent` 类的**私有方法**，无法在 `Parent` 类的**外部通过** `Parent` **类的引用调用** `Child` **类的** `priFinalMethod` **方法。**

5.3 内部类

5.3.1 内部类的基本语法

内部类的分类



例

```java
class Outer {
    public class InnerTool { // 内部类
        public int add(int a, int b) {
            return a + b;
        }
    }
    private InnerTool tool = new InnerTool();

    public void add(int a, int b, int c) {
        tool.add(tool.add(a, b), c);
        System.out.println(tool.add(tool.add(a, b), c));
    }
}
```

```java
public class Tester {
    public static void main(String args[]) {
        Outer o = new Outer();
        o.add(1, 2, 3);
        Outer.InnerTool tool = new Outer().new InnerTool();
    }
}
```

实例内部类

创建实例内部类的实例

在创建实例内部类的实例时，外部类的实例必须已经存在，例如要创建InnerTool类的实例，必须先创建Outer外部类的实例
两种语法：

- `Outer.InnerTool tool=new Outer().new InnerTool();`
- `Outer outer=new Outer();`
  `Outer.InnerTool tool =outer.new InnerTool();`
  以下代码会导致编译错误：`Outer.InnerTool tool=new Outer.InnerTool();`

实例内部类访问外部类的成员

- 在内部类中，可以**直接访问外部类的所有成员**，包括成员变量和成员方法。
- 实例内部类的实例**自动持有外部类的实例的引用。**

例

```java
public class A {
    private int a1;
    public int a2;
    static int a3;
    public A(int a1, int a2) {
        this.a1 = a1;
        this.a2 = a2;
    }
    protected int methodA() {
        return a1 * a2;
    }
    class B{  //内部类
        int b1=a1;  //直接访问private的a1
        int b2=a2;  //直接访问public的a2
        int b3=a3;  //直接访问static的a3
        int b4=new A(3,4).a1;  //访问一个新建的实例A的a1
        int b5=methodA();  //访问methodA()方法
      }
      public static void main(String args[]){
        A.B b=new A(1,2).new B();
        System.out.println("b.b1="+b.b1);  //打印b.b1=1
        System.out.println("b.b2="+b.b2);  //打印b.b2=2
        System.out.println("b.b3="+b.b3);  //打印b.b3=0
        System.out.println("b.b4="+b.b4);  //打印b.b4=3
        System.out.println("b.b5="+b.b5);  //打印b.b5=2
      }
}
```

静态内部类

- 静态内部类的实例**不会自动持有外部类的特定实例的引用**
- 在创建内部类的实例时，**不必创建外部类的实例。**

```java
class AA {
    public static class B {
        int v;
    }
}

class Tester {
    public void test() {
        AA.B b = new AA.B();
        b.v = 1;
    }
}
```

- 客户类可以通过**完整的类名直接访问静态内部类的静态成员。**

```java
public class AAA {
    public static class B{
        int v1;
        static int v2;
        public static class C{
          static int v3;
          int v4;
        }
      }
}
```

```java
public class TesterAAA {
    public void test() {
        AAA.B b = new AAA.B();
        AAA.B.C c = new AAA.B.C();
        b.v1 = 1;
        b.v2 = 1;
        // AAA.B.v1=1; //编译错误
        AAA.B.v2 = 1; // 合法
        AAA.B.C.v3 = 1; // 合法
    }
}
```

局部内部类

- 局部内部类只能在**当前方法**中使用。
- 局部内部类和实例内部类一样，可以**访问外部类的所有成员**
- 此外，局部内部类还可以访问**函数中的最终变量或参数**(final)

例

```java
public class AAAA {
    int a;
    public void method(final int p1, final int p2) {
        final int localV1 = 1;
        final int localV2 = 2;
        int localV3 = 0;
        localV3 = 1; // 修改局部变量
        class B {
            int b1 = a; // 合法，访问外部类的实例变量
            int b2 = p1; // 合法，访问final类型的参数
            int b3 = p2; // 合法，访问final类型的参数
            int b4 = localV1; // 合法,访问实际上的最终变量
            int b5 = localV2; // 合法，访问最终变量
            // int b6=localV3; //编译错误，localV3不是最终变量或者实际上的最终变量
        }
    }
}
```

5.3.2 内部类的用途

- 封装类型：如果**一个类只能由系统中的某一个类访问**，可以定义为该类的内部类。
- 直接访问外部类的成员
- 回调外部类的方法

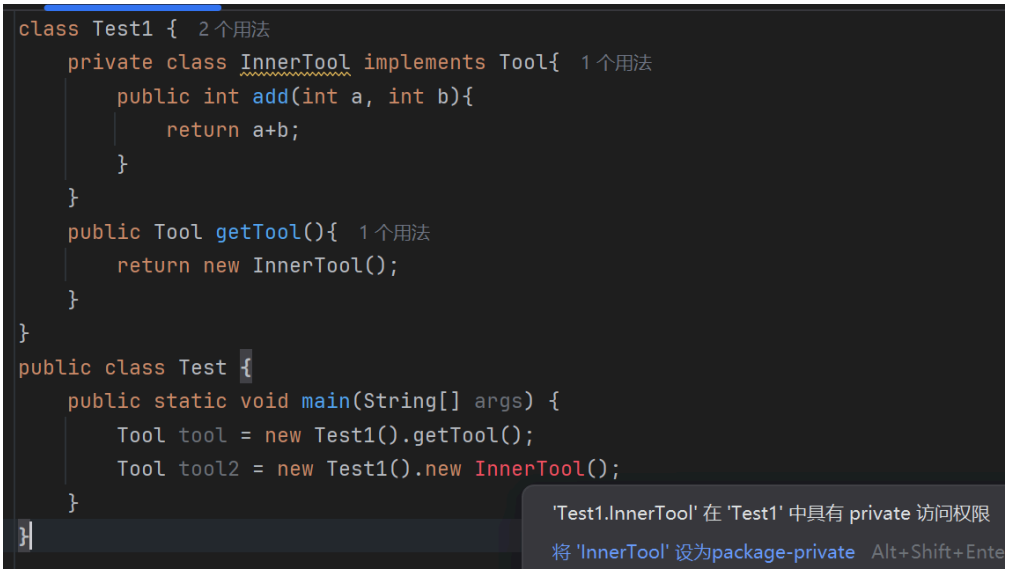
5.3.2.1 内部类封装类型

- 顶层类只能处于**public和默认**访问级别
- 而成员内部类可以处于public、protected、默认和private四个访问级别。
- 此外，如果一个**内部类仅仅为特定的方法提供服务**，那么可以把这个**内部类定义在方法之内。**

例

```
public interface Tool {
    public int add(int a, int b);
}

public class Outer {
    private class InnerTool implements Tool {
        public int add(int a, int b) {
            return a + b;
        }
    }
    public Tool getTool() {
        return new InnerTool();
    }
    public static void main(String[] args){
        //InnerTool实例向上转型为Tool类型
        Tool tool=new Outer().getTool();
        Tool tool1=new Outer().new InnerTool();
    }
}
```

- 虽然 `InnerTool` 是 `Test` 的私有内部类，但它仍然可以在 `Test` 类的内部被访问。在 `main` 方法中，`new Test().new InnerTool()` 是在 `Test` 类的内部创建 `InnerTool` 的实例，因此这是允许的。
- 在客户类中不能访问Outer.InnerTool类，但是可以通过Outer类的getTool()方法获得InnerTool的实例



5.3.2.2 内部类访问外部类的成员

- 内部类的一个特点是能够**访问外部类的各种访问级别的成员。**
- 假定有类A和类B，类B的reset()方法负责重新设置类A的实例变量count的值。一种实现方式是把类A和类B都定义为外部类

```
class A{
  private int count;
  public int add(){return ++count;}

  public int getCount(){return count;}

  public void setCount(int count){
       this.count=count;}
}
```

```
class B{
  A a;  //类B与类A关联
  B(A a){this.a=a;}
  public void reset(){
    if(a.getCount()>0)
      a.setCount(1);
    else
      a.setCount(-1);
  }
}
```

- 假如需求中**要求类A的count属性不允许被除类B以外的其他类读取或设置**，那么以上实现方式就不能满足这一需求。
- 在这种情况下，把类B定义为内部类就可以解决这一问题，而且会使程序代码更加简洁

```
class A{
  private int count;
  public int add(){return ++count;}

  class B{ //定义内部类B
    public void reset(){
      if(count>0)
        count=1;
      else
        count=-1;
    }
  }
}
```

5.3.2.3 回调

在以下Adjustable接口和Base类中都定义了adjust()方法，这两个方法的参数签名相同，但是有着不同的功能。

```
public interface Adjustable{
 /** 调节温度 */
 public void adjust(int temperature);
}
public class Base{
 private int speed;
 /** 调节速度 */
 public void adjust(int speed){
  this.speed=speed;
 }
}
```


- 如果有一个Sub类同时具有调节温度和调节速度的功能，那么Sub类需要继承Base类，并且实现Adjustable接口，但是以下代码并不能满足这一需求：

adjust(int speed)
调节速度

```
public class Sub extends Base implements Adjustable{
 private int temperature;
 public void adjust(int temperature){
  this.temperature=temperature;
 }
}
```

adjust(int temerature)
调节温度

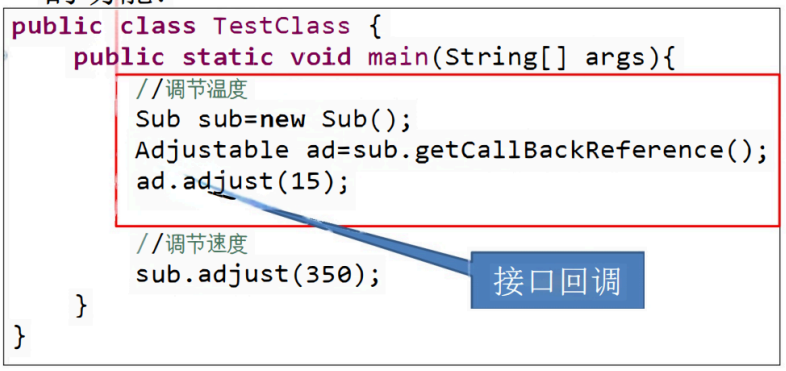
故考虑使用回调方法

回调实质上是指**一个类(Sub)尽管实际上实现了某种功能(调节温度)，但是没有直接提供相应的接口，客户类可以通过这个类的内部类(Closure)的接口(Adjustable)来获得这种功能。而这个内部类本身并没有提供真正的实现，仅仅调用外部类的实现(adjustTemperature)。**

可见，回调充分发挥了内部类具有访问外部类的实现细节的优势。

```
public class Sub extends Base {
  private int temperature;

  private void adjustTemperature(int temperature){
    this.temperature=temperature;
  }

  private class Closure implements Adjustable{
    public void adjust(int temperature){
      adjustTemperature(temperature);
    }
  }
  public Adjustable getCallBackReference(){
    return new Closure();
  }
}
```

- 以下代码演示客户类使用Sub类的调节温度的功能：



5.3.3 内部类的文件命名

对于每个内部类，Java编译器会生成独立的.class文件。这些类文件的命名规则如下：

- 成员内部类：外部类的名字$内部类的名字
- 局部内部类：外部类的名字$数字和内部类的名字

- 匿名类：外部类的名字$数字

```
public class AAAAAA {
    static class B {
    } // 成员内部类，对应A$B.class
    class C { // 成员内部类，对应A$C.class
        class D {
        } // 成员内部类，对应A$C$D.class
    }
    public void method1() {
        class E {
        } // 局部内部类1，对应A$1E.class

        B b = new B() {
        }; // 匿名类1，对应A$1.class
        C c = new C() {
        }; // 匿名类2，对应A$2.class
    }
    public void method2() {
        class E {
        } // 局部内部类2，对应A$2E.class
    }
}
```

Java编译器编译以上程序，会生成以下类文件：

A.class

A$B.class

A$C.class

ACD.class

A$1E.class

A$1.class

A$2.class

A$2E.class

5.4 匿名类

- 匿名类就是没有名字的类，是将类和类的方法定义在一个表达式范围里。
- 匿名类本身**没有构造方法**，但是会**调用父类的构造方法。**
- 匿名内部类将内部类的定义与生成实例的语句合在一起，并省去了类名以及关键字“class”,”extends”和“implements”等
- 匿名类必须继承自一个具体的**类**或实现一个**接口。**

例1

```
public class AAAAA {
    AAAAA(int v) {
        System.out.println("another constructor");
    }
    AAAAA() {
        System.out.println("default constructor");
    }
    void method() {
        System.out.println("from AAAAA");
    };
    public static void main(String args[]) {
        new AAAAA().method(); // 打印from AAAAA
        AAAAA a = new AAAAA() { // 匿名类
            void method() {
                System.out.println("from anonymous");
            }
        };
        a.method(); // 打印from anonymous
    }
}
```

```
default constructor
from AAAAA
default constructor
from anonymous
```

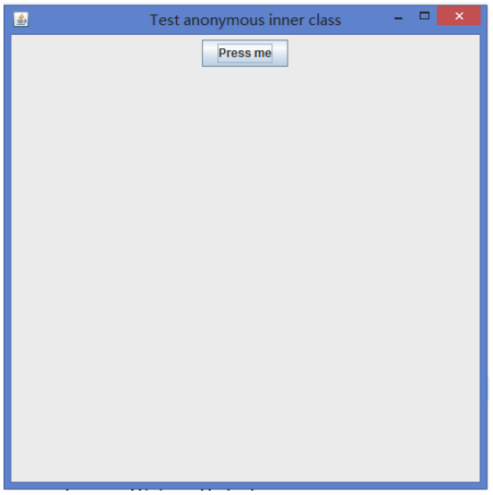
例2

```
import javax.swing.*; // 导入Swing包
import java.awt.*; // 导入AWT包
import java.awt.event.*; // 导入事件处理包

public class J_Test1 extends JFrame { // 定义J_Test1类，继承自JFrame
    public J_Test1() { // 构造函数
        super("Test anonymous inner class"); // 调用父类构造函数，设置窗口标题
        Container container = getContentPane(); // 获取内容面板
        container.setLayout(new FlowLayout(FlowLayout.CENTER)); // 设置布局为流式布局，居中对齐
        JButton b = new JButton("Press me"); // 创建按钮
        container.add(b); // 将按钮添加到内容面板
        b.addActionListener( // 为按钮添加事件监听器
            new ActionListener() { // 匿名内部类实现ActionListener接口
                public void actionPerformed(ActionEvent e) { // 实现actionPerformed方法
                    System.out.println("The button is pressed"); // 打印消息
                }
            }
        );
        setSize(100, 80); // 设置窗口大小
```

```
        setVisible(true); // 设置窗口可见
    }

    public static void main(String[] args) { // main方法
        J_Test1 application = new J_Test1(); // 创建J_Test1对象
        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //
设置窗口关闭操作
    }
}
```

```
import javax.swing.*; // 导入Swing包
import java.awt.*; // 导入AWT包
import java.awt.event.*; // 导入事件处理包

public class J_Test2 extends JFrame { // 定义J_Test2类，继承自JFrame
    public J_Test2() { // 构造函数
        super("Test anonymous inner class"); // 调用父类构造函数，设置窗口标
题
        Container container = getContentPane(); // 获取内容面板
        container.setLayout(new FlowLayout(FlowLayout.CENTER)); // 设置
布局为流式布局，居中对齐
        JButton b = new JButton("Press me"); // 创建按钮
        container.add(b); // 将按钮添加到内容面板
        b.addActionListener(new J_ActionListener()); // 为按钮添加事件监听
器，使用外部类J_ActionListener
        setSize(100, 80); // 设置窗口大小
        setVisible(true); // 设置窗口可见
    }

    public static void main(String[] args) { // main方法
        J_Test2 application = new J_Test2(); // 创建J_Test2对象
        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //
设置窗口关闭操作
    }
}

class J_ActionListener implements ActionListener { // 定义
J_ActionListener类，实现ActionListener接口
    public void actionPerformed(ActionEvent e) { // 实现actionPerformed
方法
        System.out.println("The button is pressed"); // 打印消息
    }
}
```

输出结果



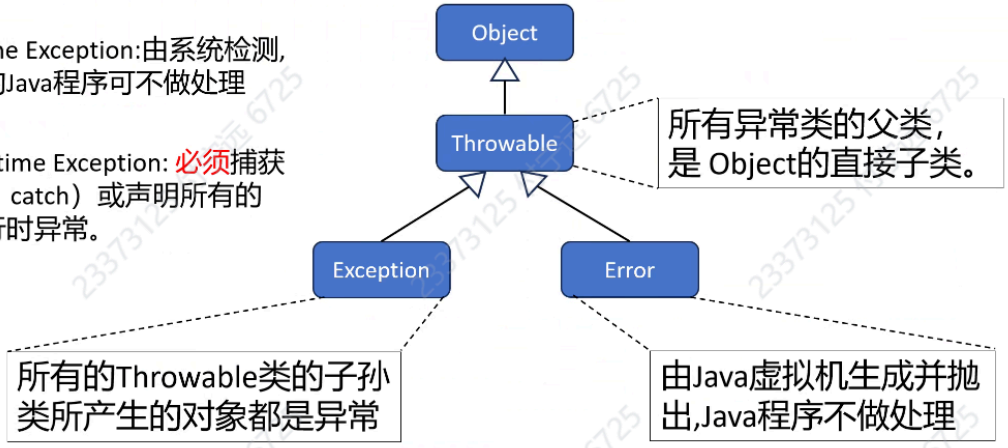
其他 9 11-14

异常处理 9

异常处理

Runtime Exception:由系统检测，用户的Java程序可不做处理

非Runtime Exception: 必须捕获（try，catch）或声明所有的非运行时异常。

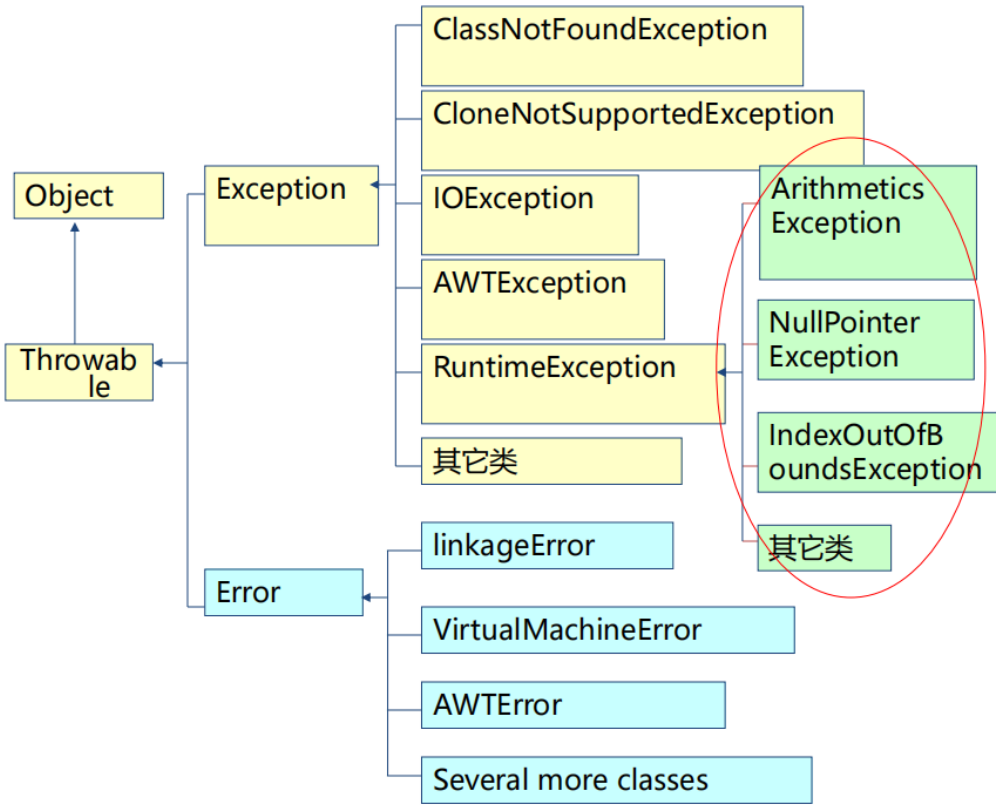


1 异常概述

- 3类错误
  - 编译错误
  - 逻辑错误
  - **运行时错误**：在程序运行过程中如果发生了一个不可能执行的操作，就会出现运行时错误。
- **异常**：一个**可以正确运行**的程序在**运行中**可能发生的错误。
- 异常特点：偶然性、可预见性、严重性
- **异常处理（Exception Handling）**：提出或者是研究一种机制，能够较好的处理程序不能正常运行的问题。

2 java异常类/异常的层次结构

java异常类



- Throwable：所有异常类的父类，是Object的直接子类。
- Error：由Java虚拟机生成并抛出，Java程序不做处理
- Exception：所有的Throwable类的子孙类所产生的对象都是异常
  - Runtime Exception：**编译时不可监测的异常**，由系统检测，用户的Java程序可不做处理，系统将它们交给缺省的异常处理程序。
  - **非Runtime Exception**：**编译时可以监测的异常**，Java编译器要求Java程序**必须**捕获或声明所有的非运行时异常，可以通过try-catch或throws处理。
- throw：用户自己产生异常。

常见的异常

1. ArithmeticException
2. ArrayIndexOutOfBoundsException
3. ArrayStoreException
4. IOException
5. FileNotFoundException
6. NullPointerException
7. MalformedURLException
8. NumberFormatException
9. OutOfMemoryException

异常分类

- 非受检异常 (unchecked exception) ： Runtime Exception 及其子类、Error 及其子类。
  - 只能在程序执行时被检测到，**不能在编译时被检测到**；
  - 程序可不处理，交由系统处理。
- 受检异常 (checked exception)：除了非受检异常之外的异常（即其他的异常类都是可检测的类）
  - 这些异常在**编译时**就能被java编译器所检测到异常。
  - 必须采用 **throws 语句**或者 **try-catch** 方式处理异常

3 java异常处理机制

- 抓抛模型
  - 抛出(throw)异常：Java程序在正常的执行过程中，一旦出现异常，就会在异常代码处生成一个对应**异常类的对象**，并将此对象抛出，且其后代码就**不再执行。**
    - 关于异常对象的产生
      - 系统**自动**生成异常对象
      - **手动**生成一个异常对象，并抛出（**throw**）
  - 捕获(catch)异常：可以理解为异常处理方式。
    - try-catch-finally
    - throws
- Java语言按照面向对象的思想来处理异常：
  - 把各种不同类型的异常情况进行分类，用Java类来表示异常情况，这种类被称为异常类。
  - 用throws语句在方法声明处声明抛出特定异常。（**只抛出不处理，交给调用该方法的方法进行处理，若一直不处理，则交给系统处理**）
  - 用try-catch语句来捕获并处理异常。（**处理**）
  - 用throw语句在方法中抛出具体的异常。（**自定义异常**）

4 try-catch-finally

`finally`：无条件执行的语句

- `try{}` 中执行 `return` ， `finally` 语句仍然执行，在 `return` 前执行

- `try{}` 中执行 `exist(0)`，`finally` 语句不执行

```
package buaa.com.exceptionEx;
public class Test {
    public static void main(String[] args){
        System.out.println("main中x="+Test.test());
    }
    static int test(){
        int x=1;
        try{
            return x;
        }
        finally{
            ++x;
            System.out.println("finally中x="+x);
        }
    }
}
```

`try-catch-finally` 语句格式

- 一个try一个catch可以，一个try一堆catch也可以，try-catch-finally也可以，try-fianlly也可以

- `try-catch-finally` 语句的语法格式
  - 一般 `finally` 写**释放资源**的部分（打开水龙头后无论水龙头能不能使最后都要关上水龙头）
  - 如果一个异常类和其子类都出现在catch子句中，应把**子类放在前面**，否则将永远不会到达子类。

```
try {
        // 接受监视的程序块,在此区域内发生的异常,由catch中指定的程序处理;
}
catch (ExceptionType1 e) {
        // 抛出ExceptionType1异常时要执行的代码
}
catch (ExceptionType2 e) {
        // 抛出ExceptionType2异常时要执行的代码
}……
[finally {
        // 无条件执行的语句
}]
```

- `try-finally`

```
//try-finally
try{
        // 对文件进行处理的程序
}
finally{
        // 不论是否发生异常，都关闭文件
}
```

- `catch(异常名1|异常名2|异常名3 变量)`

```
try{

}catch(异常名1 |异常名2 |异常名3… 变量）{

}
```

方法虽简洁，但是也不是特别完美

- 上述异常必须是**同级**关系；
- **处理方式是一样**的（针对同类型的问题，给出同一个处理）

try-with-resourse

- 资源：所有实现Closeable的类，如流操作，socket操作，httpclient等
- 打开的资源越多，finally中嵌套的将会越深，所以引入了 Try-with-resourse
- 带资源的try语句（try-with-resource）的最简形式为：

```
try(Resource res = xxx){ // 可指定多个资源
        work with res
}
```

- 处理规则
  - 凡是实现了AutoCloseable接口的类，在try()里声明该类实例的时候，在try结束后，**close方法都会被调用**，这个动作会**早于finally里调用的方法。**
  - 不管是否出现异常，try()里的实例都会被调用；
  - close方法**越晚声明**的对象，会**越早被close**掉。

例1

先声明的后关闭

```
public class Main2 {
    public static void main(String[] args) {
        try(ResourceSome some = new ResourceSome();
             ResourceOther other = new ResourceOther()) {
            some.doSome();
            other.doOther();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

class ResourceSome implements AutoCloseable {
    void doSome() {
        System.out.println("do something");
    }
    @Override
    public void close() throws Exception {
        System.out.println("some resource is closed");
    }
}

class ResourceOther implements AutoCloseable {
    void doOther() {
        System.out.println("do other things");
    }
    @Override
    public void close() throws Exception {
        System.out.println("other resource is closed");
    }
}
```

```
do something
do other things
other resource is closed
some resource is closed
```

5 Throws

用于**声明异常**。

- 声明异常：一个方法**不处理**它产生的异常，而是**沿着调用层次向上传递**，由调用它的方法来处理这些异常，叫声明异常。若最终方法也没有处理异常，异常将**交给系统处理**
- Throws语句用来表明一个方法可能抛出的各种异常，并说明该方法会**抛出但不捕获**异常

- 声明异常的格式

```
<访问权限修饰符><返回值类型><方法名>(参数列表) throws 异常列表{}
```

- **当父类中的方法没有throws，则子类重写此方法时也不可以throws。**若重写方法中出异常，必须采用try结构处理。
- 重写方法不能抛出比被重写方法**范围更大**的异常类型，子类重写方法也**可以不抛出**异常。

```
// 方法**不处理**它产生的异常，而是**沿着调用层次向上传递**，由调用它的方法来处理这些异常
class ThrowsTest4 {
    // 声明异常，但不处理异常
    static void method() throws IllegalAccessException {
        System.out.println("\n在 method 中抛出一个异常");
        throw new IllegalAccessException();
    }
    public static void main(String[] args) {
        try {
            method();
        } catch (IllegalAccessException e) {
            System.out.println("在 main 中捕获异常: " + e);
        }
    }
}
//
// 在 method 中抛出一个异常
// 在 main 中捕获异常: java.lang.IllegalAccessException
```

```
// 若最终方法也没有处理异常，异常将交给系统处理
import java.io.FileInputStream;
import java.io.IOException;

public class Test {
    public static void main(String[] args) throws IOException {
        FileInputStream in = new FileInputStream("myfile.txt");
        int b;
        while ((b = in.read()) != -1) {
            System.out.print((char) b);
        }
        in.close();
    }
}
// Exception in thread "main" java.io.FileNotFoundException:
myfile.txt (系统找不到指定的文件。)
//      at java.base/java.io.FileInputStream.open0(Native Method)
//      at
```

```
java.base/java.io.FileInputStream.open(FileInputStream.java:213)
//      at java.base/java.io.FileInputStream.<init>
(FileInputStream.java:152)
//      at java.base/java.io.FileInputStream.<init>
(FileInputStream.java:106)
//      at Test.main(Test.java:6)
```

例

```
public class MainCatcher{
  public void methodA(int money)throws SpecialException{
    if(--money<=0) throw new SpecialException("Out of money");
    System.out.println("methodA");
  }
  public void methodB(int money) throws SpecialException{
    methodA(money);
    System.out.println("methodB");
  }
  public static void main(String args[]){
    try{
      new MainCatcher().methodB(1);
      System.out.println("main");
    }catch(SpecialException e){
      System.out.println("Wrong");
    }}}
```

该程序的打印结果为：Wrong。

```
public class MainCatcher{
  public void methodA(int money)throws SpecialException{
    if(--money<=0) throw new SpecialException("Out of money");
    System.out.println("methodA");
  }
  public void methodB(int money) throws SpecialException{
    methodA(money);
    System.out.println("methodB");
  }
  public static void main(String args[]){
    try{
      new MainCatcher().methodB(2);
      System.out.println("main");
    }catch(SpecialException e){
      System.out.println("Wrong");
    }}}
```

该程序的打印结果为：
methodA
methodB
main

```
public class WithFinally {
  public void methodA(int money)throws SpecialException{
    if(--money<=0)
      throw new SpecialException("Out of money");
    System.out.println("methodA");
  }
  public static void main(String args[]){
     try{
       new WithFinally().methodA(1);
       System.out.println("main");
     }catch(SpecialException e){
       System.out.println("Wrong");
     }finally{
       System.out.println("Finally");
     }}}
```

该程序的打印结果为：
Wrong
Finally

```
public class WithFinally {
  public void methodA(int money)throws SpecialException{
    if(--money<=0)
       throw new SpecialException("Out of money");
    System.out.println("methodA");
  }
  public static void main(String args[]){
     try{
       new WithFinally().methodA(2);
       System.out.println("main");
     }catch(SpecialException e){
       System.out.println("Wrong");
     }finally{
       System.out.println("Finally");
     }}}
```

该程序的打印结果为：
methodA
main
Finally

6 throw与创建自定义异常类

- throw抛出**用户自定义异常。**
- 用户定义的异常必须由**用户自己抛出** `<throw><异常对象>` `throw new MyException`
- 程序会在throw语句处立即终止，转向 try…catch 寻找异常处理方法。
- 语句格式

```
<class><自定义异常名>extends<Exception>{
        public String to string(){
                return "myException";
        }
}
```

```java
public class MyExceptionTest {
    public static void main(String args[]) {
        try {
            System.out.println("\n进入监控区，执行可能发生异常的程序段");
            method(8);
            method(20);
            method(6);
        } catch (MyException e) {
            System.out.println("\t程序发生异常并在此处进行处理");
            System.out.println("\t发生的异常为：" + e.toString());
        }
        System.out.println("这里可执行其他代码");
    }
    static void method(int a) throws MyException {
        System.out.println("\t此处引用 mathod (" + a + ")");
        if (a > 10)
            throw new MyException(a);// 主动抛出MyException
        System.out.println("正常返回");
    }
}
```

```java
class MyException extends Exception {  //自定义异常
    private int x;
    MyException(int a) {
        x = a;
    }
    public String toString() {
        return "MyException";
    }
}
```

```
进入监控区，执行可能发生异常的程序段
        此处引用 mathod (8)
正常返回
        此处引用 mathod (20)
        程序发生异常并在此处进行处理
        发生的异常为：MyException
这里可执行其他代码
```

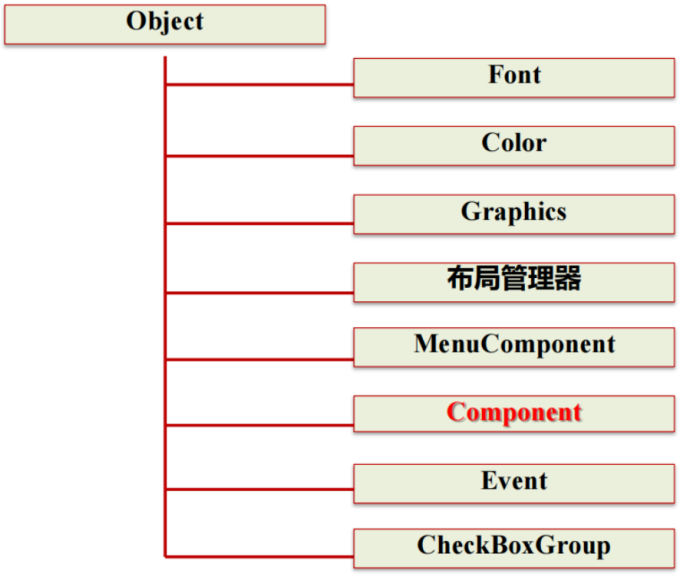
图形界面 11

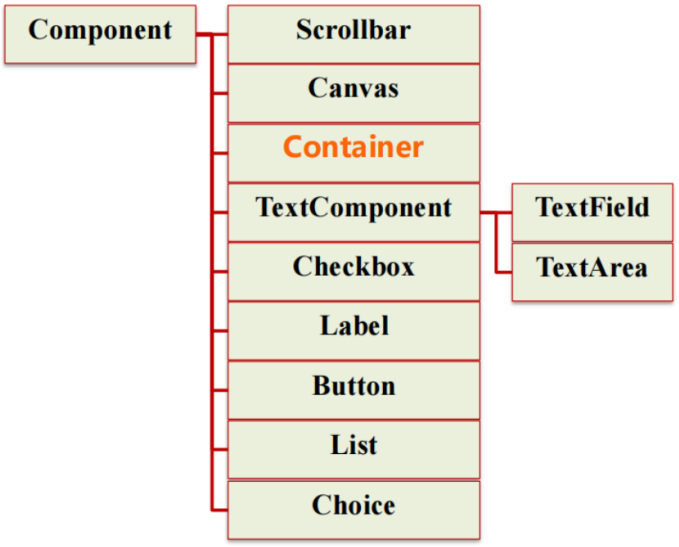
创建Java图形界面的程序类库主要有两个:

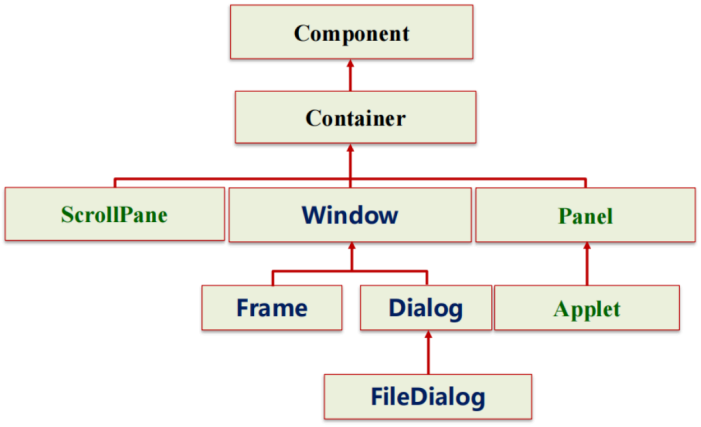
- AWT（java.awt.*）；
- Swing（javax.swing.*）

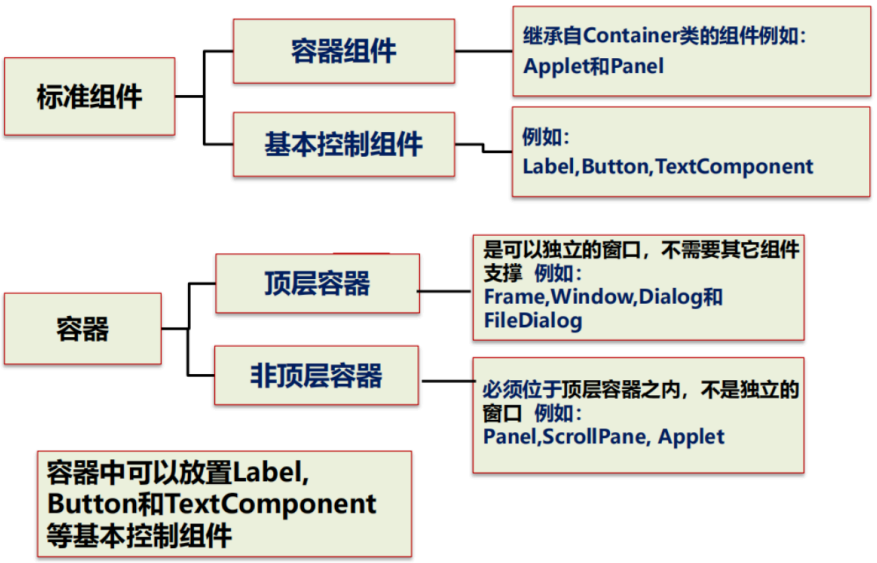
AWT

目前Java图形界面中主要使用的是AWT中的**布局管理器**和**事件处理**









容器类（Container）组件

Container类的常用方法

- add(Component comp);
- remove(Component comp);
- setLayout(LayoutManager mgr);
- setLocation(int x, int y);
- setSize(int x, int y);
- setBackground(Color color);
- show();
- setVisible(boolean b);
- validate();

//Ensures that this component has a valid layout.

控制组件

- 控制组件主要用来提供人机交互的基本控制界面
- **主要包括：**
  - 按钮（Button）
  - 单选按钮(RadioBox)
  - 复选框(Checkbox)
  - 下拉列表框(Choice)
  - 画板(Canvas)
  - 列表框(List)
  - 标签(Label)
  - 文本组件(TextField和TextArea)
  - 滚动条(ScrollBar)
  - 菜单(Menu)等

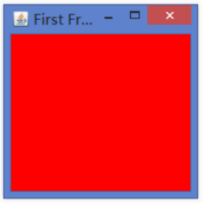
设置GUI应用程序的流程

- 引用需要的包和类
- 设置一个顶层的容器
- 根据需要为容器设置布局管理器或使用默认布局管理器
- 将组件添加到容器内，位置自行设计
- 为响应事件的组件编写事件处理代码

例

Java awt 中创建窗口

```
import java.awt.*;
import java.awt.event.*;
public class TestFrame {
    public static void main(String[] s) {
        Frame f = new Frame("First Frame");
        f.setLocation(100, 100);
        f.setSize(200, 200);
        f.setBackground(Color.red);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```



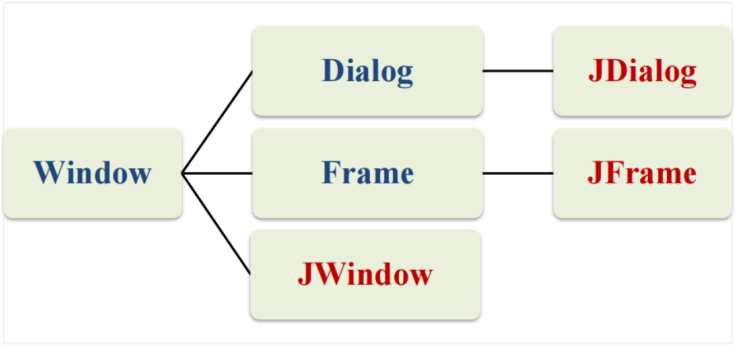

```
import java.awt.*;
import java.awt.event.*;
public class UseTextArea1{
  public static void main(String args[]){
        MyFrame f=new MyFrame();
        f.setSize(300,200);
        f.setBackground(Color.lightGray);
        f.setLayout(new FlowLayout());
        f.show();
  }
}
```

```
class MyFrame extends Frame implements ActionListener{
        TextArea ta1;          TextField tf1;
        Button btn1,btn2,btn3;
        public MyFrame(){
          super("my frame");
          ta1=new TextArea(4,30);
          tf1=new TextField(30);
          btn1=new Button("添加文本");
          btn2=new Button("插入文本");
          btn3=new Button("替换选中文本");
          btn1.addActionListener(this);
          btn3.addActionListener(this);
          btn2.addActionListener(this);
          System.out.println(this.toString());
          setLayout(new FlowLayout());
          add(ta1); add(tf1); add(btn1); add(btn2); add(btn3);
          this.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
                              dispose();
                              System.exit(0);
          }});
        }
```


```
  public void actionPerformed(ActionEvent e){
    if(e.getSource()==btn1){
            ta1.append(tf1.getText());
            tf1.setText("");
    }
    if(e.getSource()==btn2){
            ta1.insert(tf1.getText(),ta1.getCaretPosition());
            tf1.setText("");
    }
    if(e.getSource()==btn3){
      int k0=ta1.getSelectionStart();
      int k1=ta1.getSelectionEnd();
            ta1.replaceRange(tf1.getText(),k0,k1);
    }
  }
}
```

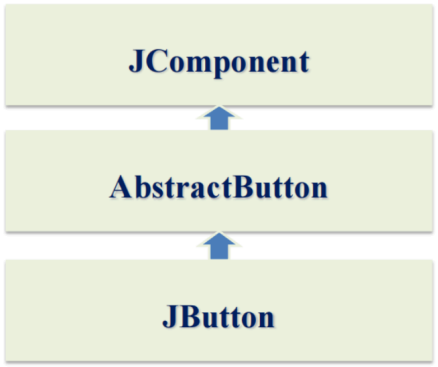
Swing

- 几乎所有AWT组件对应有新功能更强的Swing组件；
- 另外还加入了一些全新的组件；
- Swing组件在名称前面多了一个字母“J”；
- 除此之外，使用Swing开发的图形界面更美观，功能更强大。
- 在javax.swing包中，定义了两种类型的组件：
  - 顶层容器（JFrame、JApplet、JDialog和JWindow）
  - 和轻量级组件
- Swing组件从AWT的Container类继承而来。

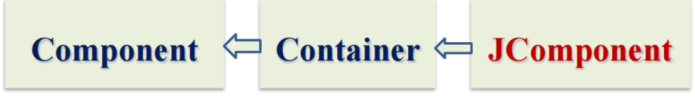


4个Swing类直接派生自其相应的AWT类，**它们是Swing的4个顶级容器**

- 除上述4个顶级容器外，其他所有组件都扩充自JComponent类。例如：



- AWT组件和Swing组件的继承关系



JFrame容器

- JFrame是放置其他 Swing 组件的顶级容器
- JFrame用于在 Swing 程序中创建窗体
- 它的构造函数：
  - **JFrame()**
  - **JFrame(String title)**
- 组件必须添加至内容面板（content pane）上，而不是直接添加至 JFrame 对象，示例：
  - frame.getContentPane().add(b);

JFrame窗体类

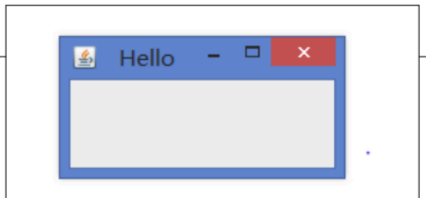
- JFrame窗体类有一个构造方法JFrame(String title)，通过它可以创建一个以参数为标题的JFrame对象。
- 当JFrame被创建后，它是不可见的，必须通过以下方式使JFrame成为可见的：
  - 先调用setSize(int width,int height)显式设置JFrame的大小，或者调用pack()方法自动确定JFrame的大小，pack()方法会确保JFrame容器中的组件都会有与布局相适应的合理大小。
  - 然后调用setVisible(true)方法使JFrame成为可见的

设置关闭窗体的操作

- JFrame的setDefaultCloseOperation(int operation)方法用来决定如何响应用户关闭窗体的操作，参数operation有以下可选值：
  - JFrame.DO_NOTHING_ON_CLOSE：什么也不做。
  - JFrame.HIDE_ON_CLOSE：隐藏窗体，这是JFrame的默认选项。
  - JFrame.DISPOSE_ON_CLOSE：销毁窗体。
  - JFrame.EXIT_ON_CLOSE：结束程序。

创建一个简单窗体

```
import javax.swing.*;

public class SimpleFrame1 {
    public static void main(String args[]) {
        JFrame jFrame = new JFrame("Hello");
        jFrame.setSize(200, 100); // 设置JFrame的宽和高
        jFrame.setVisible(true); // 使JFrame变为可见
    }
}
```



设置GUI应用程序的流程

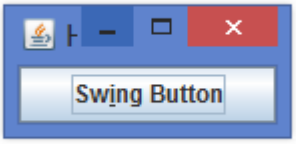
- 引用需要的包和类
- 设置一个顶层的容器
- 根据需要为容器设置布局管理器或使用默认布局管理器
- 将组件添加到容器内，位置自行设计
- 为响应事件的组件编写事件处理代码

注意事项

- 在创建GUI时，必须让轻构件完全包含在顶级Swing构件中，当然也可以包含在其他轻构件中，但必须有JDialog、JFrame、JWindow作为根容器

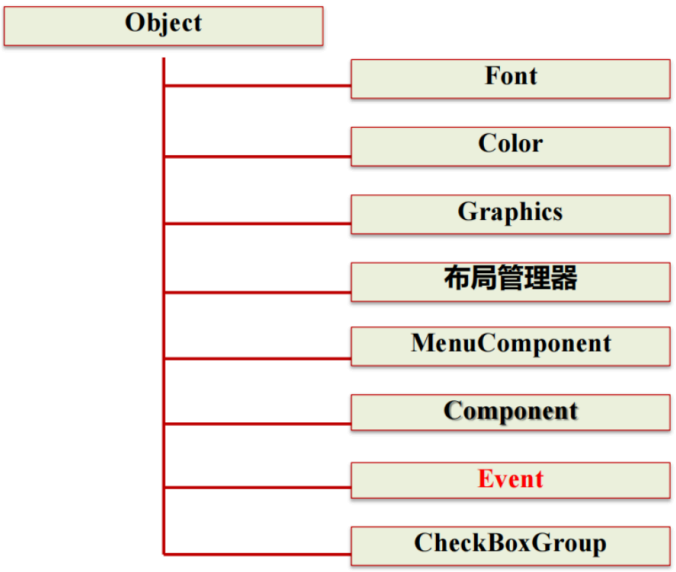
创建一个包含按钮的窗体

```
import javax.swing.*;
public class SimpleFrame2 {
    public static void main(String[] args) {
    JFrame myFrame=new JFrame("Hello");
    JButton myButton = new JButton("Swing Button");

    // 创建一个快捷键：用户按下Alt-i键等价于点击该Button
    myButton.setMnemonic('i');

    //设置鼠标移动到该Button时的提示信息
    myButton.setToolTipText("Press me");

    myFrame.add(myButton);

    //当用户选择myFrame窗体的关闭图标，将结束程序
    myFrame.setDefaultCloseOperation(myFrame.EXIT_ON_CLOSE);
    myFrame.pack();
    myFrame.setVisible(true);
  }
}
```

事件处理

package java.awt



- Java AWT和Swing组件共用了**java.awt.event.***类和接口来处理事件（event）
- 所以涉及到事件处理时，应引入**java.awt.event.***

Java事件处理机制

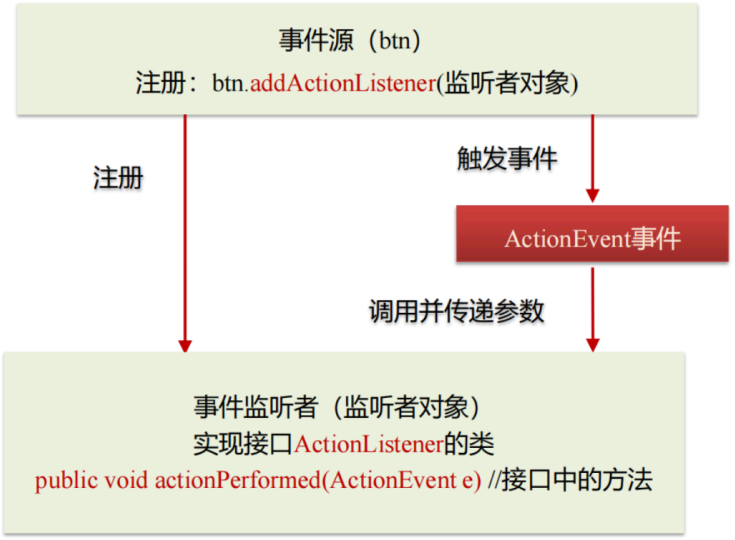
- Java的事件处理使用的是“委托事件模型”
  - 事件：GUI中用户交互行为所产生的一种效果
  - 事件源：GUI中每个可能产生事件的组件
  - 事件处理（监听）者：接收事件并进行处理的方法。
  - *Java中所有的组件都从Component类中继承了将事件处理授权给监听者的方法*

事件处理（监听）者

- 在Java的委托事件模型中，事件处理者可以是任何类的对象，*只要这个类实现了一个监听者接口，那么这个对象就可以处理事件。*也就是事件监听者具有监听和处理某类事件的功能
  - 在此，又一次体现了接口是一种能力

事件处理机制

- 事件处理模型分为3个部分：事件源对象、事件监听器对象与事件对象；
- 能产生事件的组件叫做事件源，如按钮；
- 事件监听器注册在事件源对象（例如按钮或包含按钮的容器）上，用来监听事件是否发生，当事件发生时将调用事件处理方法解决问题；
- 事件对象用来封装已发生的事件的信息，在事件发生后，将信息传递给事件处理者，事件处理者中的相应方法处理事件；



Java事件处理(1)：注册监听者

- 在每一个事件处理者的程序中都需要编写3段代码
- （1）在生成了组件后（例如按钮），确定该事件源要响应的事件（一个或者多个），注册相应的事件监听者（一个或者多个）。
  - 例如:
  - someComponent.addActionListener (instance of MyClass);

Java事件处理（2）：编写监听者类

- （2）：在事件处理者的类声明中，编写代码说明该类实现了一个监听者接口，或继承了一个实现了监听者接口的类。如：

```
class MyClass implements ActionListener  {
… …
}
```

Java事件处理（3）：实现了监听者接口的方法

- （3）在事件处理者的类中，都有实现了监听者接口的方法，在该方法中，编写业务逻辑，处理事件。如：

```
public void actionPerformed(ActionEvent  e)
{…
 …
}
```

```
public class J_Test2 extends JFrame{
    public J_Test2() {
        super("Test anonymous inner class");
        Container container = getContentPane();
        container.setLayout(new FlowLayout(FlowLayout.CENTER));
        JButton b = new JButton("Press me");
        container.add(b);
        b.addActionListener(new MyClass());
        setSize(100, 80);
        setVisible(true);
    }
    public static void main(String[] args) {
        J_Test1 application = new J_Test1();
        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    class MyClass implements ActionListener{
       public void actionPerformed(ActionEvent e) {
            System.out.println("The button is pressed");
        }
    }
}
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class J_Test1 extends JFrame {
    public J_Test1() {
        super("Test anonymous inner class");
        Container container = getContentPane();
        container.setLayout(new FlowLayout(FlowLayout.CENTER));
        JButton b = new JButton("Press me");
        container.add(b);
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("The button is pressed");
            }
        });
        setSize(100, 80);
        setVisible(true);
    }
    public static void main(String[] args) {
        J_Test1 application = new J_Test1();
        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

主要的接口类型及其作用

ActionListener	按钮、菜单、双击列表
AdjustmentListener	滚动条
ComponentListener	控件状态改变
ContainerListener	控件加入或删除
FocusListener	焦点得失
ItemListener	复选框、列表、可选菜单
KeyListener	键盘输入
MouseListener	鼠标输入
MouseMotionListener	鼠标拖拽
TextListener	文本区文本改变
WindowListener	Window状态改变

Frame类处理窗口关闭事件

- 当一个窗口打开、关闭、最小化时都会引发窗口事件（WindowEvent），实现窗口事件监听的接口是WindowListener接口
- 如果一个窗口通过实现WindowListener接口来处理窗口事件，则需要实现接口中的7个方法

Frame类处理窗口关闭事件

WindowListener接口中包含以下7个方法：

```
public void windowActivated(WindowEvent e);
public void windowClosed(WindowEvent e);
public void windowClosing(WindowEvent e);
public void windowDeactivated(WindowEvent e);
public void windowDeiconified(WindowEvent e);
public void windowIconified(WindowEvent e);
public void windowOpened(WindowEvent e);
```

```
import java.awt.*;  import java.awt.event.*;
public class TestFrame {
public static void main(String[] s){
   Frame f = new Frame("First Frame");
   f.setLocation(100, 100);
   f.setSize(200,200);
   f.setBackground(Color.red);
   f.setVisible(true);
   f.addWindowListener(new WindowAdapter(){
 public void windowClosing(WindowEvent e){
                System.exit(0);}});
}}
```

事件适配器

- 为方便起见，Java为某些监听者接口提供了适配器类（XXXAdapter），当需要对某种事件进行处理时，只需要让事件处理类继承事件所对应的适配器类，重写需要关注的方法即可，而不必实现无关的方法

- 事件适配器包括以下几种
  - –ComponentAdapter
  - –ContainerAdapter
  - –FocusAdapter
  - –KeyAdapter
  - –MouseAdapter
  - –MouseMotionAdapter
  - –WindowAdapter

Event-adapter class	Implements interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

由于`WindowAdapter`是`WindowListener`接口的一个适配器类，它提供了`WindowListener`接口中所有方法的空实现，因此我们只需要重写我们关心的事件处理方法，即`windowClosing`。其他`WindowListener`接口中的方法（如`windowOpened`、`windowClosed`等）不需要实现，因为`WindowAdapter`已经提供了默认的空实现。

布局管理器

Java里有六种布局管理器

- FlowLayout(顺序布局)
- BorderLayout（边界布局）
- GridLayout（网格布局）
- BoxLayout
- CardLayout（卡片布局）
- GridBagLayout(网格包布局)
- null布局

FlowLayout布局

- 每个部件从左到右、从上到下，依据容器的大小逐行在容器中顺序摆放
- FlowLayout是Applet类和Panel类、JPanel类的默认布局方式

- FlowLayout中的主要方法
  - 构造函数

    FlowLayout();

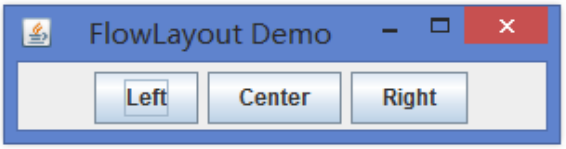
    FlowLayout(int align);

    FlowLayout(int align, int hgap, int vgap);
  - 设置布局

    setLayout(new FlowLayout());

对齐方式
FlowLayout.RIGHT
FlowLayout.LEFT
FlowLayout.CENTER

表示组件之间间隔



BorderLayout布局

- BorderLayout布局方式是将组件按东、南、西、北、中五种方位放置在容器中
- 如果东南西北某个位置上没有放置组件，则该区域会被中间区域和相关的某个位置区域自动充满
- BorderLayout是Frame、JFrame和Dialog、JApplet的默认布局方式

BorderLayout布局

- 构造函数

1. BorderLayout();
2. BorderLayout(int hgap, int vgap)

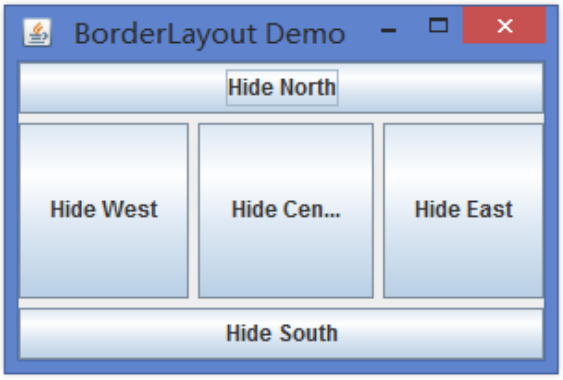
- 设置布局

  setLayout(new BorderLayout());

- 添加组件到指定布局

  addLayoutComponent(Component comp,Object constraints);

BoredeLayout.EAST
BoredeLayout.SOUTH
BoredeLayout.WEST
BoredeLayout.NORTH
BoredeLayout.CENTER



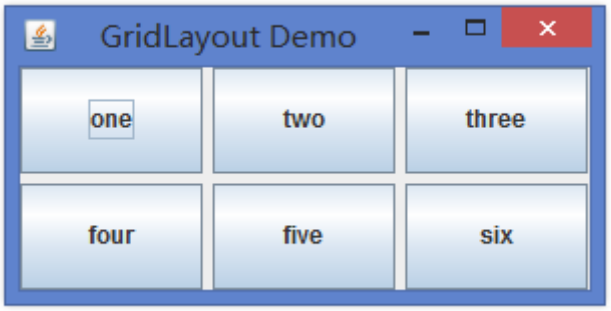
GridLayout布局

- GridLayout布局
  - 将每个组件放置在rows行及columns列中，即将容器分成大小相等的矩形域，当一排满了，就从下一行开始
- GridLayout的构造函数

```
GridLayout(int rows, int cols);
GridLayout(int rows, int cols, int hgap, int vgap);
```

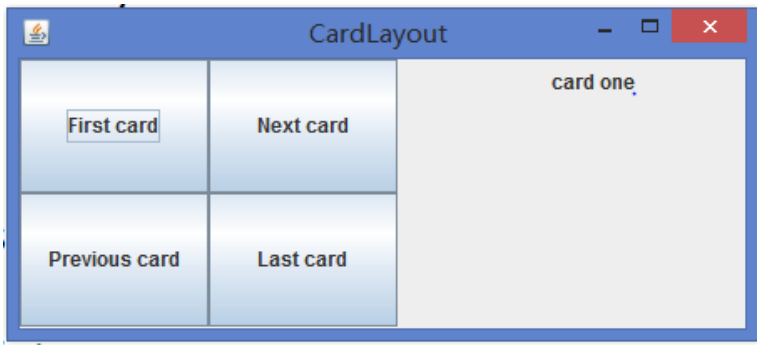
- 设置布局

```
setLayout(new GridLayout(3,3,5,5));
```



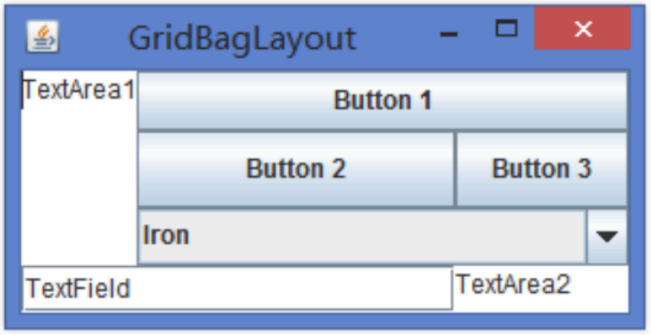
CardLayout布局

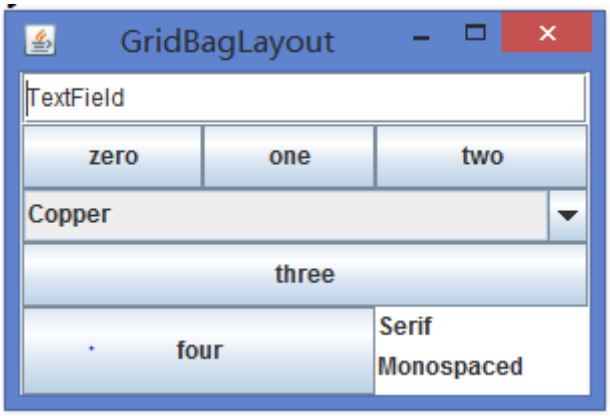
- CardLayout布局管理是把容器的所有组件当成一叠卡片，卡片布局方式中只有其中的一个组件，即一张卡片被显示出来，其余组件是不可见的
  - 构造函数
    - CardLayout(int hgap, int vgap);
    - CardLayout();
- 常用方法
  - addLayoutComponent(Component comp,Object constraints);
  - first(Container parent);
  - last(Container parent);
  - next(Container parent);
  - show(Container parent, String name);



GridBagLayout布局

- GridBagDemo.java





null布局

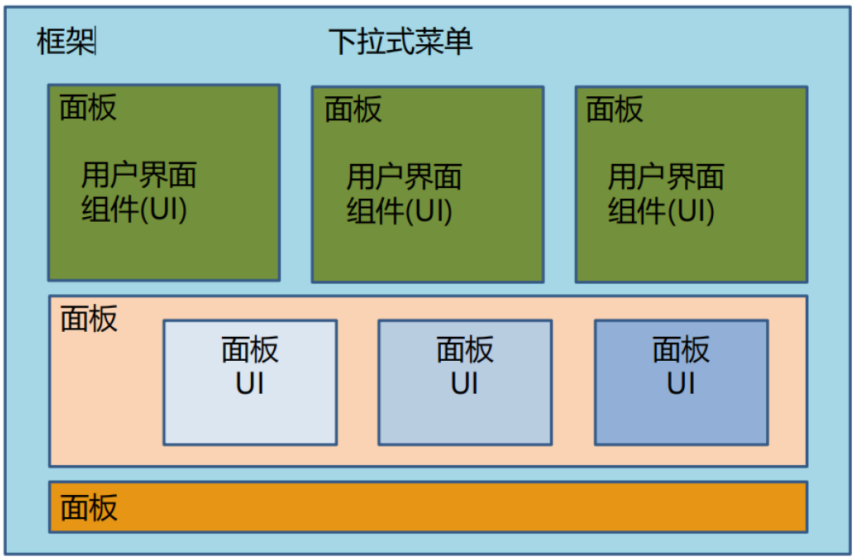
- null布局又称为空布局
- 对一个容器（Container）而言，可以用下面方式设置其布局管理器为null
  - **public void setLayout(null);**
- 这样，容器内的组件（Component）可以利用下面的方法来设置其大小和位置
  - **public void setBounds(int x, int y,int width,int height);**

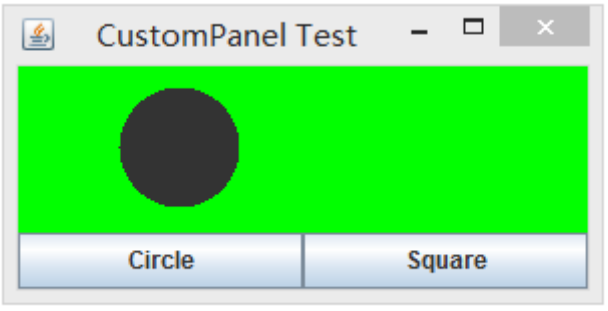
x,y表示组件左上角在容器中的坐标，width和height表示组件的宽和高

容器的嵌套

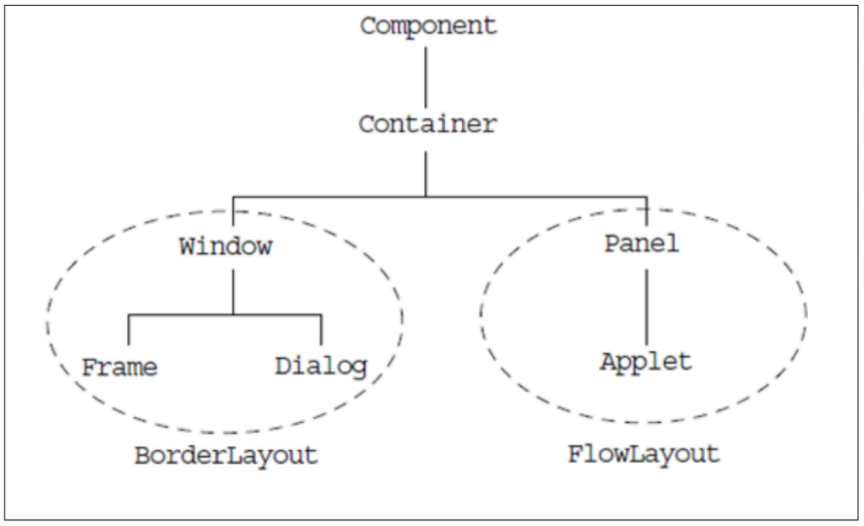
- 容器的嵌套：在实际的编程中，我们经常遇到向容器中添加容器，这就是容器的嵌套
- **JPanel类型的容器常常扮演这种角色，在多种布局方式之间起到了一种桥梁的作用**

- 面板容器JPanel
  - JPanel 组件是一个中间容器
  - 用于将小型的轻量级组件组合在一起
  - JPanel 的缺省布局为 FlowLayout


框架
下拉式菜单
面板
用户界面
组件(UI)
面板
用户界面
组件(UI)
面板
用户界面
组件(UI)
面板
面板
UI
面板
UI
面板
UI
面板


CustomPanel Test
Circle
Square

默认布局管理器


Component
Container
Window
Panel
Frame
Dialog
Applet
BorderLayout
FlowLayout

对话框

对话框（JDialog）

- JDialog表示对话框。对话框是在现有窗口的基础上弹出的另一个窗口。
- 对话框可用来处理个别细节问题，使得这些细节不与原先窗口的内容混在一起。

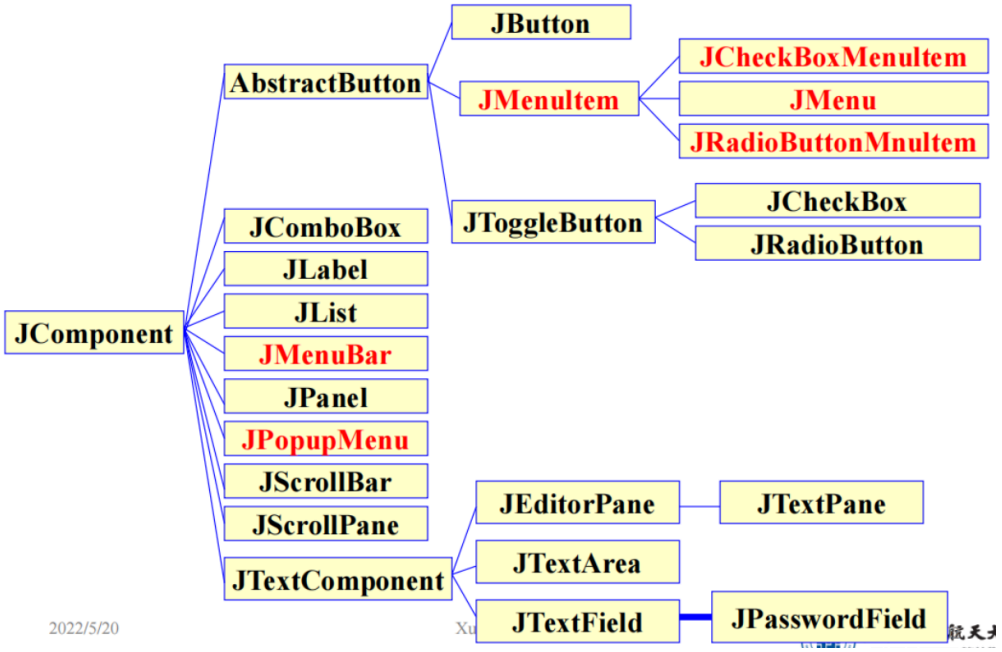
- 对话框(JDialog)是与框架类似的有边框、有标题、可移动的独立存在的容器，默认的布局方式为BorderLayout布局
- 对话框不能作为程序的最外层容器，也不能包含菜单栏，它被框架所拥有并由框架负责弹出
- 默认的对话框是不显示的，需用setVisible();
- 对话框分为模态对话框和非模态对话框两种

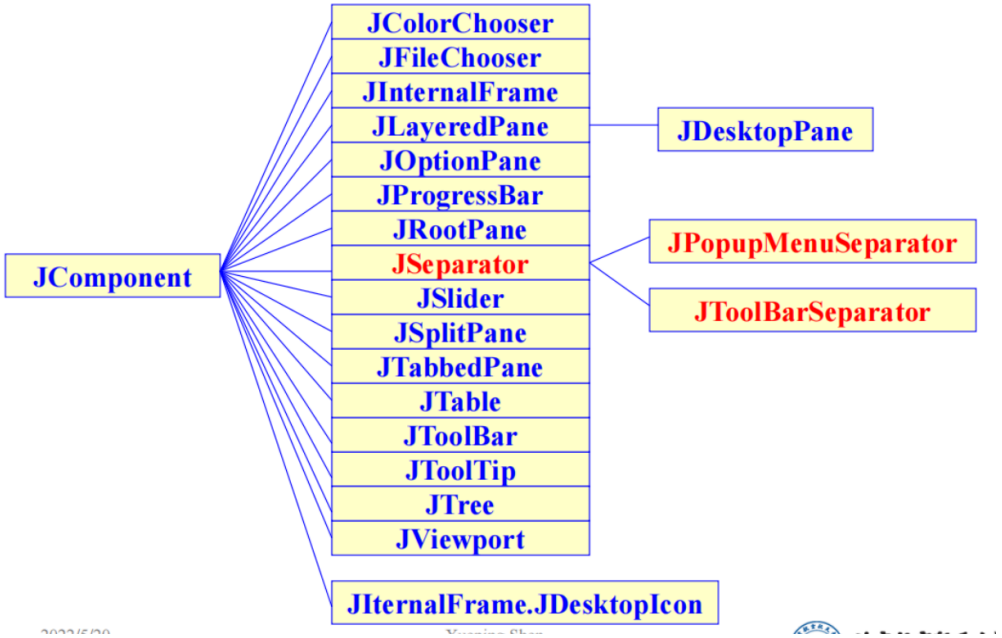
模态对话框、非模态对话框、文件对话框

- 模态对话框：即对话框显示时程序被阻塞，在对话框被关闭之前，其他窗口无法接受任何形式的输入。
- 非模态对话框：没有上述限制
- **文件对话框：**是用于文件选择的对话框，允许用户对目录或文件进行浏览和选择。

- 对话框（JDialog）具有以下形式的构造方法：
  - public JDialog(Frame owner,String title,boolean modal)
  - owner表示对话框所属的Frame，参数title表示对话框的标题，参数modal有两个可选值：
  - 参数modal为true：表示模式对话框，这是JDialog的默认值。
  - 参数modal为false：表示非模式对话框。
- 当对话框被关闭时，通常不希望结束整个应用程序，因此只需调用JDialog的dispose()方法销毁对话框，从而释放对话框所占用的资源。

菜单

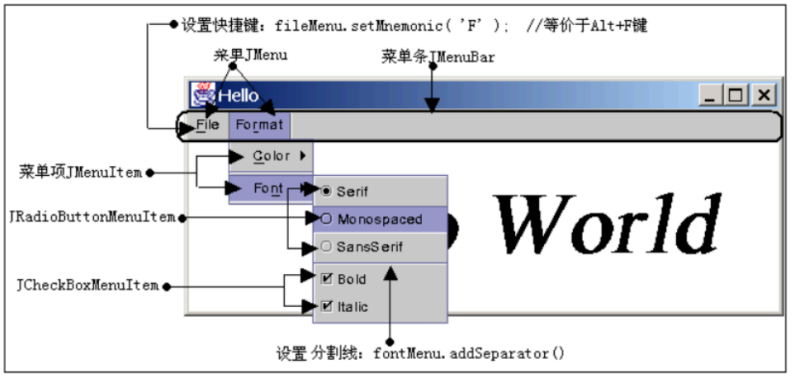

JComponent
AbstractButton
JButton
JMenuItem
JCheckBoxMenuItem
JMenu
JRadioButtonMnuItem
JToggleButton
JCheckBox
JRadioButton
JComboBox
JLabel
JList
JMenuBar
JPanel
JPopupMenu
JScrollBar
JScrollPane
JTextComponent
JEditorPane
JTextPane
JTextArea
JTextField
JPasswordField


JComponent
JColorChooser
JFileChooser
JInternalFrame
JLayeredPane
JDesktopPane
JOptionPane
JProgressBar
JRootPane
JSeparator
JPopupMenuSeparator
JSlider
JToolBarSeparator
JSplitPane
JTabbedPane
JTable
JToolBar
JToolTip
JTree
JViewport
JInternalFrame.JDesktopIcon

Java 菜单的组织方式

- 支持菜单的组件有JFrame、JDialog和Japplet。这些组件中有一个setMenuBar(JMenuBar bar)方法，可以用这个方法来设置菜单条。
- 一个菜单条JMenuBar中可以包含多个菜单Jmenu
- 一个菜单JMenu中可以包含多个菜单项JMenuItem。
- 当用户选择了某个菜单项，就会触发一个ActionEvent事件，该事件由ActionListener负责处理。

- JMenuItem有两个子类：JRadioButtonMenuItem和JCheckBoxMenuItem，它们分别表示单选菜单项和复选菜单项。

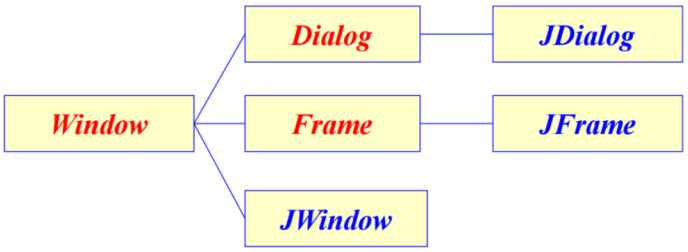


小节

小结(事件处理)

- 确认触发的事件，取得事件类（如ActionEvent）的名字，并删掉其中的“Event”，加入“Listener”；
- 实现上面的接口，针对想要捕获的事件编写方法代码；
- 为事件处理器（监听者接口）创建一个对象，让自己的组件和方法完成对它的注册。

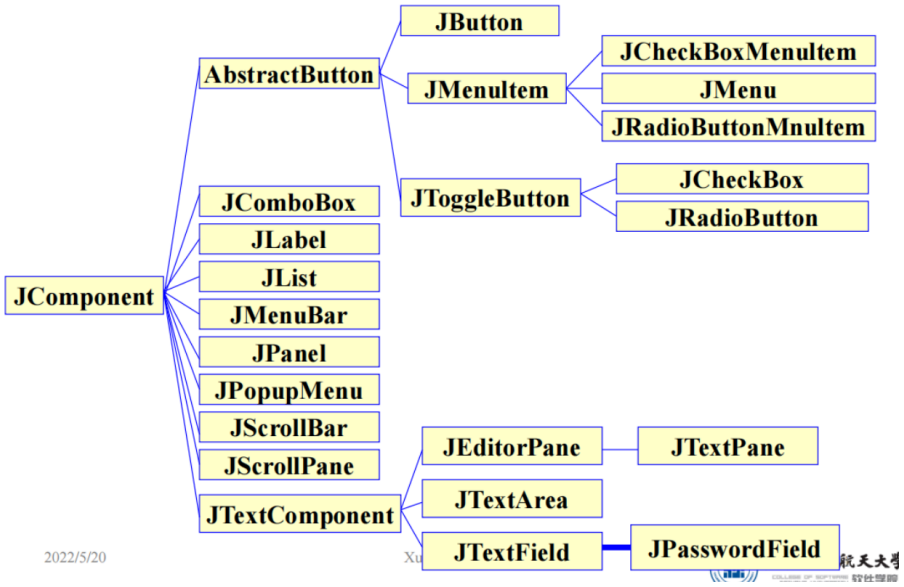
小结：*JFrame and Windows*



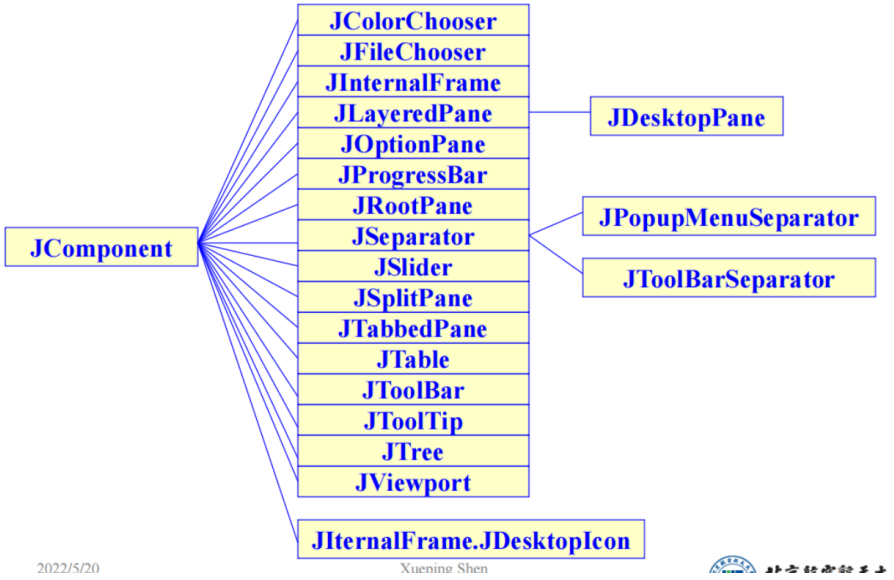
小结：创建容器与组件基本步骤

- 创建顶层容器（常用的为窗口对象）
- 创建内容面板，设置其背景颜色，设置其布局管理器
- 创建普通面板，设置其背景颜色，设置其位置、大小，设置其布局管理器
- 创建组件，设置其背景颜色，设置其位置、大小、字体等
- 将面板添加到窗口，将组件添加到指定面板
- 创建事件监听器，实现事件接口方法
- 给事件源注册监听器

小结：*Component Hierarchy—AWT Similar*



小结：*ComponentHierarchy—New and Expanded Components*



Java IO 12

scanner

- Scanner的作用：通过分隔符模式将输入分解为标记，默认情况下该分隔符模式与空白匹配。
- 通过Scanner 类的 `next()` 与 `nextLine()` 方法获取输入的字符串，在读取前我们一般需要使用 `hasNext` 与 `hasNextLine` 判断是否还有输入的数据
- **Scanner**

```
Scanner scanner = new Scanner(System.in);
while(scanner.hasNextDouble()){
```

```
    double x = scanner.nextDouble();
}
```

- **转换大小写**:

```
String original = "Hello, World!";
String uppercase = original.toUpperCase();
String lowercase = original.toLowerCase();
```

- **替换字符串**:

```
String original = "Hello, World!";
String replaced = original.replace("World", "Java");
```

- **分割字符串**:

```
String original = "one,two,three";
String[] parts = original.split(",");
```

- **连接字符串**:

```
String part1 = "Hello, ";
String part2 = "World!";
String combined = part1 + part2;
```

- **格式化字符串**:

```
int number = 42;
String formatted = String.format("The answer is %d", number);
```

例

next

```
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // 从键盘接收数据

        // next方式接收字符串
        System.out.println("next方式接收：");
        // 判断是否还有输入
        if (scan.hasNext()) {
            String str1 = scan.next();
            System.out.println("输入的数据为：" + str1);
        }
        scan.close();
    }
}
```

```
$ javac ScannerDemo.java
$ java ScannerDemo
next方式接收：
runoob com
输入的数据为：runoob
```

nextLine

```
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // 从键盘接收数据

        // nextLine方式接收字符串
        System.out.println("nextLine方式接收：");
        // 判断是否还有输入
        if (scan.hasNextLine()) {
            String str2 = scan.nextLine();
            System.out.println("输入的数据为：" + str2);
        }
        scan.close();
    }
}
```

```
$ javac ScannerDemo.java
$ java ScannerDemo
nextLine方式接收：
runoob com
输入的数据为：runoob com
```

nextInt

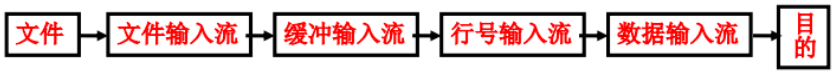
```
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // 从键盘接收数据
        int i = 0;
        float f = 0.0f;
        System.out.print("输入整数：");
        if (scan.hasNextInt()) {
            // 判断输入的是否是整数
            i = scan.nextInt();
            // 接收整数
            System.out.println("整数数据：" + i);
        } else {
            // 输入错误的信息
            System.out.println("输入的不是整数！");
        }
        System.out.print("输入小数：");
        if (scan.hasNextFloat()) {
            // 判断输入的是否是小数
            f = scan.nextFloat();
            // 接收小数
            System.out.println("小数数据：" + f);
        } else {
            // 输入错误的信息
            System.out.println("输入的不是小数！");
        }
        scan.close();
    }
}
```

```
$ javac ScannerDemo.java
$ java ScannerDemo
输入整数：12
整数数据：12
输入小数：1.2
小数数据：1.2
```

- 输入流：输入数据流只能读,不能写
  - 字节流：Java中的输入数据流(字节流)都是抽象类**InputStream**的子类；
  - 字符流：Java中的输入数据流(字符流)都是抽象类**Reader**的子类；
- 输出流：输出数据流只能写,不能读
  - 字节流:java中的输出数据流(字节流)都是抽象类**OutputStream**的子类；
  - 字符流:java中的输出数据流(字符流)都是抽象类**Writer**的子类；

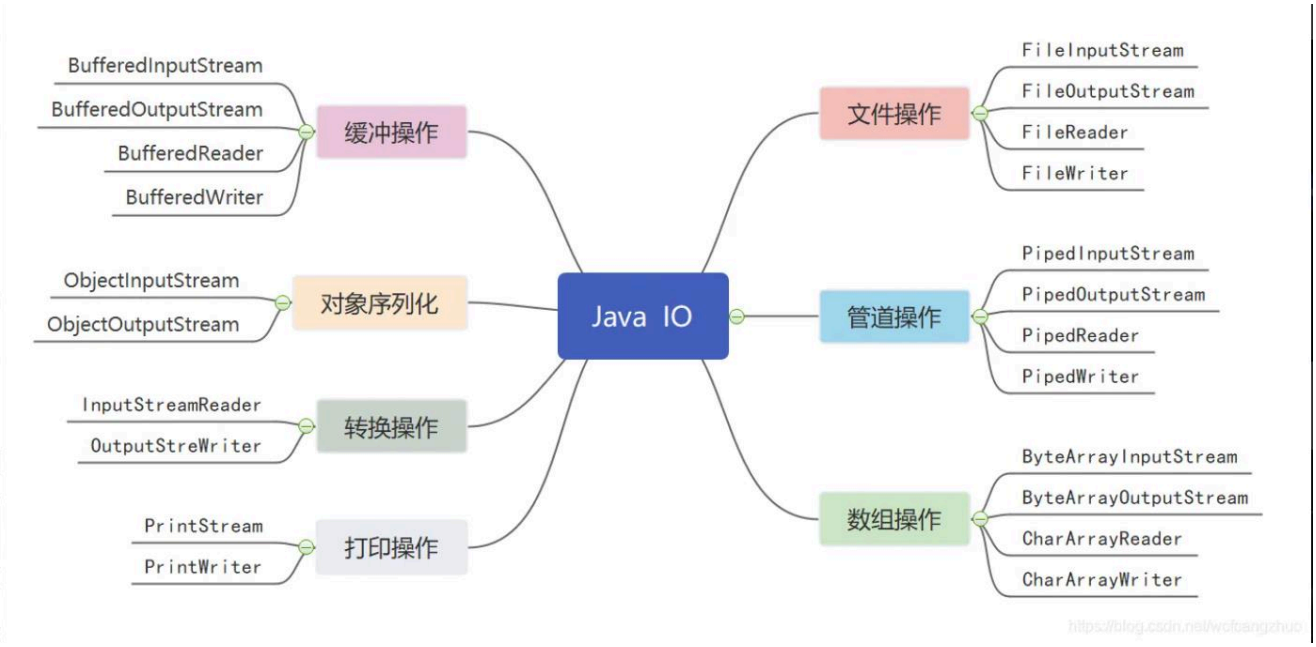
- 字节流可以操作所有类型的文件；
- 字符流只能操作**纯文本文件**；

- 在Java中有数据传输的地方都用到I/O流(通常是文件,网络,内存和标准输入输出等)
- Java 的 IO 流主要包括输入、输出两种 IO 流，每种输入、输出流有可分为字节流和字符流两大类：
  - 字节流以字节为单位来处理输入、输出操作
  - 字符流以字符为单位来处理输入、输出操作
- 众多的流对象协同工作



1. 读写的时候字节流是按字节读写，字符流按字符读写。
2. 字节流适合所有类型文件的数据传输，因为计算机字节（Byte）是电脑中表示信息含义的最小单位。字符流只能够处理纯文本数据，其他类型数据不行，但是字符流处理文本要比字节流处理文本要方便。
3. 在读写文件需要对内容按行处理，比如比较特定字符，处理某一行数据的时候一般会选择字符流。
4. 只是读写文件，和文件内容无关时，一般选择字节流。

- File, File(Input/Output)Stream, RandomAccessFile是处理本地文件的类。
- Data(Input/Output)Stream是一个过滤流的子类,借此可以读写各种基本数据, 在文件和网络中经常使用.如: readByte, writeBoolean等。
- Buffered(Input/Output)Stream的作用是在数据送到目的之前先缓存,达到一定数量时再送到目的,以提高程序的运行效率。
- Piped(Input/Output)Stream适合于一个处理的输出作为另一个处理的输入的情况。



其他 IO NIO NIO2

- IO流（同步、阻塞）
- NIO（同步、非阻塞）:NIO(NEW IO)用到块，效率比IO高很多
  三个组件：
  - Channels（通道）：**流是单向的，Channel是双向的**，既可以读又可以写，Channel可以进行异步的读写，对Channel的读写必须通过**buffer**对象
  - Buffer（缓冲区）
  - Selector（选择器）：Selector是一个对象，它可以注册到很多个Channel上，监听各个Channel上发生的事件，并且能够根据事件情况决定Channel读写。这样，通过一个线程管理多个Channel，就可以处理大量网络连接了
- NIO2(异步、非阻塞)：AIO(Asynchronous IO)

同步与异步

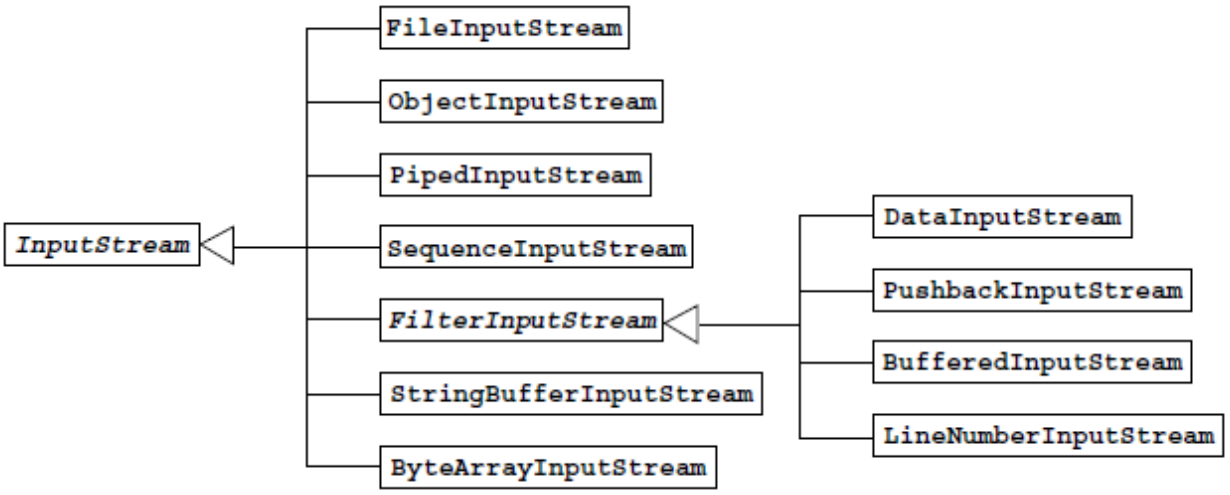
- 同步:是一种可靠的有序运行机制，进行同步操作时，后续的任务须**等待当前调用返回**，才会进行下一步；
- 异步：后续任务**不需要等待当前调用返回**，通常依靠事件、回调等机制来实现任务间次序关系；

阻塞与非阻塞

- 阻塞：在进行读写操作时，当前线程会处于阻塞状态，**无法从事其他任务**。只有当条件就绪才能继续；
- 非阻塞：不管IO操作是否结束，**直接返回**，相应操作在后台继续处理

字节流

InputStream



简介

- InputStream是**抽象类**，所以不能通过“new InputStream()”的方法构造InputStream的实例。但它声明了输入流的基本操作，**包括读取数据(read)、关闭输入流(close)、获取流中可读的字节数(available)、移动读取指针(skip)、标记流中的位置(mark)和重置读取指针(reset)** 等,它的子类一般都重写这些方法。
- 通过构造InputStream子类的实例方式可以获得InputStream类型的实例。

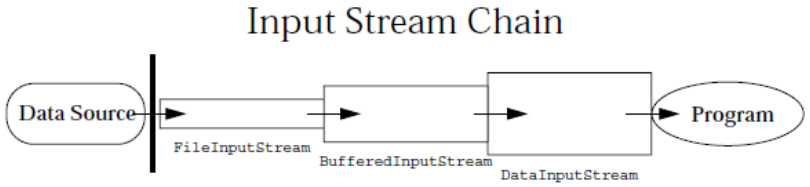
相关函数介绍

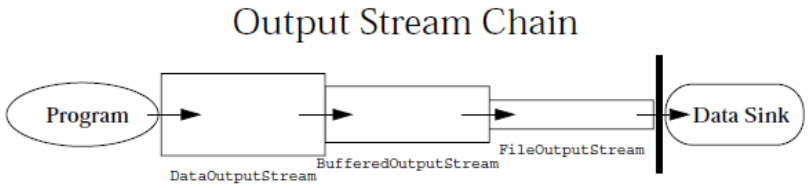
- 方法read()提供了三种从流中读数据的方法.
  - int read():一次只能读一个字节,抽象方法。
  - int read(byte b[]):一次读多个字节到数组中
  - int read(byte[],int off,int len);
- 一般available和read()混合使用，这样在读操作前可以知道有多少字符需要读入。
- mark通常与reset()方法配合使用，可重复读取输入流所指定的字节数据。

分类

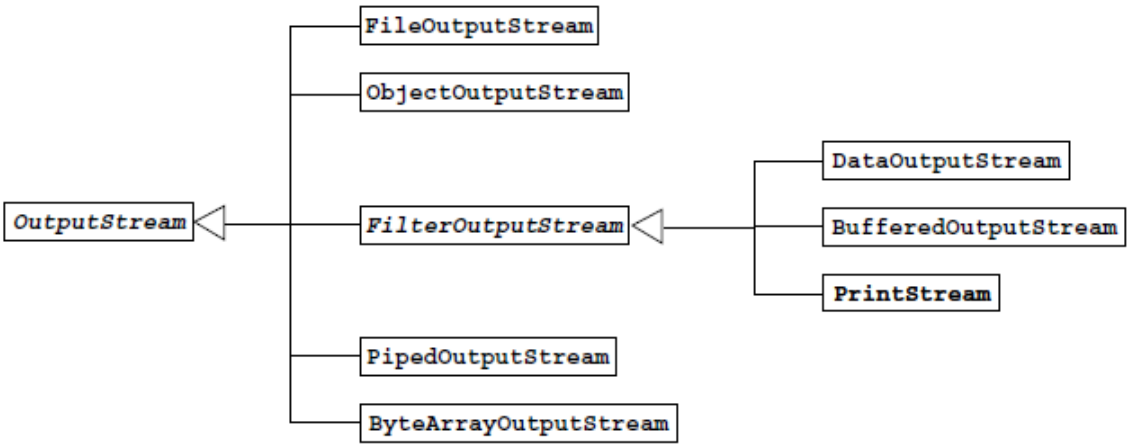
- FileInputStream：用于**从本地文件**中读出数据。
- ObjectInputStream：用来读取**对象**；要保证对象是**串行化**(Serializable)的（指**对象**通过把自己**转化为一系列字节**，记录字节的**状态数据**，以便再次利用的这个过程；**对不希望串行化的对象要用关键字transient修饰**）

- PipedIntputStream：用于管道输入/输出时从**管道**中读取数据；管道数据流的两个类一定是**成对**的，**同时使用并相互连接**的，这样才形成一个数据通信管道；管道数据流主要用于**线程间的通信。**
- SequencedInputStream：用来把**两个或更多**的InputStream输入流对象转换为单个inputStream输入流对象使用。
- FilterInputStream：提供**将流连接在一起**的能力；**某一时刻只能一个线程访问**它；其子类PushbackInputStream(**读过的一个或几个字节数据退回到输入流中/回压别的字节数据**)和BufferedInputStream读取数据时可以对数据进行**缓冲**，这样可以**提高效率**和**增加特殊功能。**
- DataInputStream实现了java.io包中的DataInput接口，读取数据的同时，可以对数据进行**格式处理。**因此，能用来读取java中的**基本数据类型。**
- ByteArrayInputStream：包含一个内存缓冲区，用于从内存中读取数据。
- AudioInputStream：Audio的输入输出





OutputStream



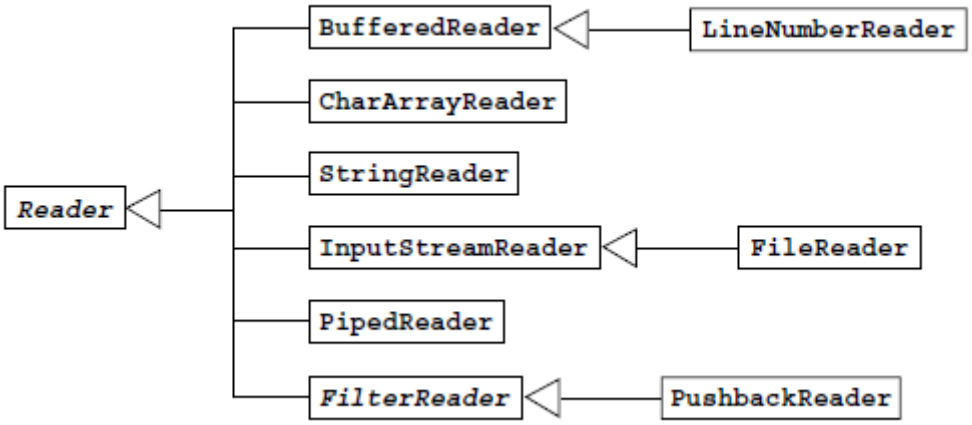
简介

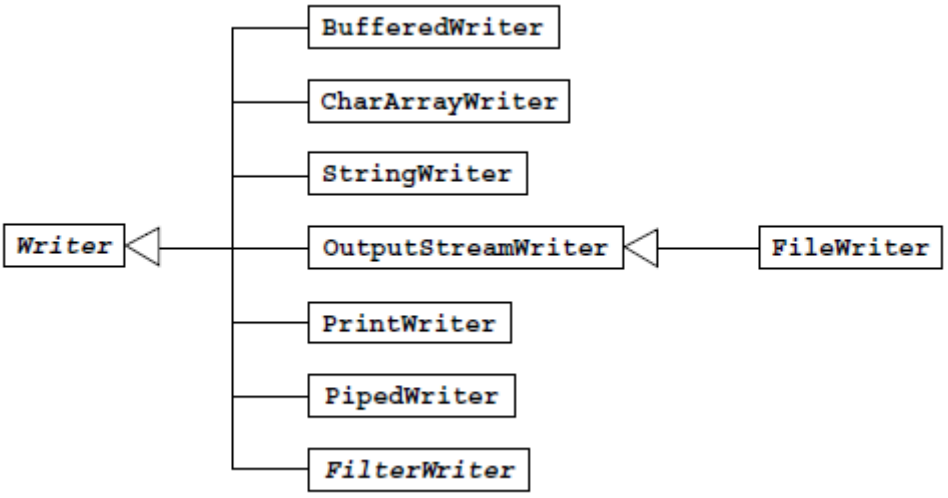
OutputStream是抽象类，所以不能通过“newOutputStream()”的方法构造OutputStream的实例。但它声明了输出流的基本操作，**包括输出数据(write)、关闭输出流(close)、清空缓冲区(flush)等**

分类

- FileOutputStream：用于向**本地文件**中写入数据。
- PipedOutputStream：用于管道输入/输出时把数据向**管道**输出
- DataOutputStream：提供了对java的**基本数据类型**的支持
- PrintStream：提供了**向屏幕输出有格式数据**的很多方法；**System.out**
  为什么PrintStream适合做打印流?
  - 它提供了更多的输出成员方法，输出的数据**不必先转换成字符串类型或其它类型。**
  - PrintStream的成员方法一般**不会抛出异常**；
  - PrintStream具有**自动强制输出(flush)功能**，即当输出回车换行时，在缓存中的数据会全部自动写入指定的文件或在标准输出窗口中显示

字符流





- Reader类和Writer类中的大部分方法与InputStream类和OutputStream类中的对应方法名称相同，只是读取或写入的数据是**字符、字符数组和字符串**等。
- 如果程序读到的数据是不同国家的语言，其编码不同，那么程序应使用Reader和Writer流。

InputStreamReader类和OutputStreamWriter类

- InputStreamReader类继承自Reader类，通过其read方法从字节输入流中读取一个或多个**字节数据转换为字符数据**，它不是一个缓冲流，因此其**转换的效率并不高。**
- OutputStreamWriter类继承自Writer类，其作用是**转变字符输出流为字节流输出。**
- InputStreamReader类和OutputStreamWriter类都可以**接一个缓冲流来提高效率**

FileReader和FileWriter

FileReader和FileWriter类分别是Reader和Writer子类，他们分别用来从**字符文件读取字符**和**向字符文件输出字符数据。**

多线程 13

多线程

BETTER!!!

Thread	Runnable
简单易用	编码较复杂
java不支持多继承，因此有局限性	可以避免单继承的限制

适合资源的共享

更推荐TimeUnit.xxxx.sleep

Sleep	yield
使线程转入阻塞状态	使线程转入runnable状态
不会考虑线程的优先级	相同优先级或更高的线程运行机会

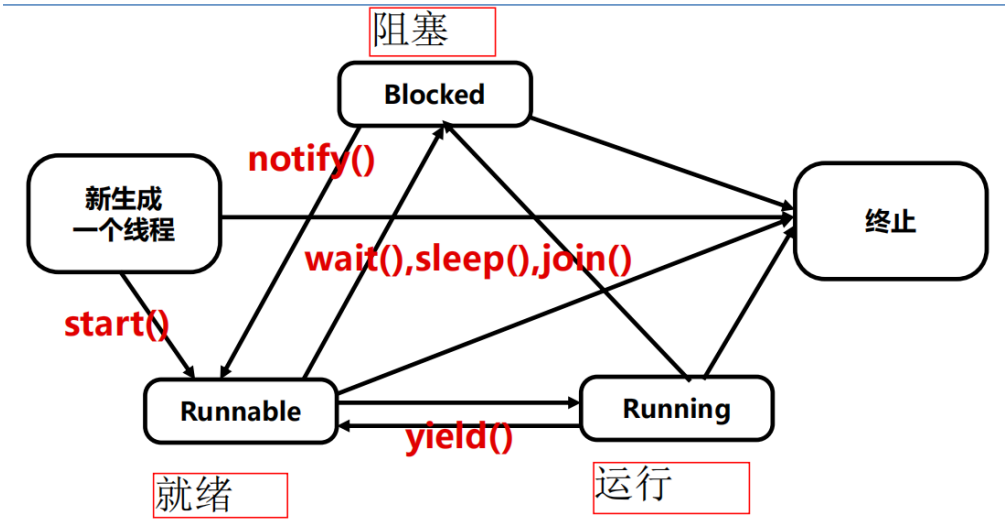
runnable局限性：run()返回值是void；不允许抛出任何已检查的异常——callable借口既有返回值又能抛出异常

wait, notify, notifyAll必须在已经持有锁的情况下执行,所以它们只能出现在**synchronized**作用的范围内，也就是出现在**synchronized**修饰的方法中。

wait的作用:**释放已持有的锁**,进入等待队列. **sleep不释放锁**

notify的作用:**随机唤醒**wait队列中的一个线程并把它移入锁申请队列

notifyAll的作用:唤醒wait队列中的所有的线程并把它们移入锁申请队列



1 进程的概念

- **程序**（program）：静态的**代码。**

- **进程**（process）是程序的一次**执行过程。**
- 程序是静态的，进程是动态的。
- 不同进程所**占用的系统资源相对独立**；
- 属于同一进程的所有线程**共享该进程的系统资源**；
- 线程本身既没有入口，也没有出口，其自身也**不能独立运行**，完成其任务后，自动终止，也可以由进程使之强制终止。

当多线程程序执行时**具有并发执行的多个线程**；

为什么用多线程？

- 速度快：线程之间**共享相同的内存单元**(代码和数据)，因此在线程间切换，不需要很大的系统开销，所以线程之间的**切换速度远远比进程之间快**，线程之间的通信也比进程通信快的多。
- CPU利用率高：多个线程轮流抢占CPU资源而运行时，从微观上讲，一个时间里只能有一个作业被执行，在宏观上可使多个作业被同时执行，即等同于要让多台计算机同时工作，使系统资源特别是**CPU的利用率得到提高**，从而可以提高**整个程序的执行效率。**

2 线程的运行

每个线程都有一个独立的程序计数器和方法调用栈（method invocation stack）：

- 栈存简单局部变量，堆存类对象
- 线程运行中需要的资源：CPU、方法区的代码、堆区的数据、栈区的方法调用栈

3 线程的调度

线程的调度

- 在Java中，线程调度通常是**抢占式**(即哪一个线程先抢到CPU资源则先运行)，而不是**分时间片式。**
- 一旦一个线程获得执行权，这个线程将**持续运行下去**，直到它运行结束或因为某种原因而阻塞，或者有另一个高优先级线程就绪（这种情况称为**低优先级线程被高优先级线程所抢占**）。
- 所有被阻塞的线程按次序排列，组成一个**阻塞队列。**
- 所有就绪但没有运行的线程则根据其优先级排入一个**就绪队列。**
- 当CPU空闲时，如果就绪队列不空，就绪队列中第一个具有最高优先级的线程将运行。
- 当一个线程被抢占而停止运行时，它的运行态被改变并放到就绪队列的队尾；
- 一个被阻塞（可能因为睡眠或等待I/O设备）的线程就绪后通常也放到就绪队列的队尾

优先级

线程的调度是按：

1. 其优先级的高低顺序执行的；
2. 同样优先级的线程遵循“先到先执行的原则”

线程优先级

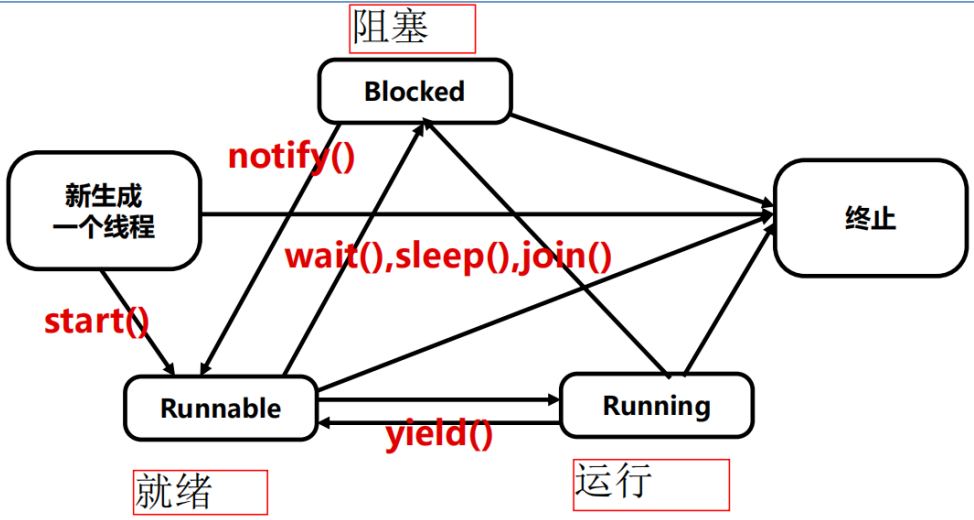
- 范围 1~10 （10 级）。数值越大，级别越高
- Thread 类定义的 3 个常数：
  - MIN_PRIORITY 最低(小)优先级（值为1）
  - MAX_PRIORITY 最高(大)优先级（值为10）
  - NORM_PRIORITY 默认优先级（值为5）
- 线程创建时，**继承父线程的优先级。**
- 常用方法：
  - getPriority( )：获得线程的优先级
  - setPriority( )：设置线程的优先级

主线程

- main( ) 方法：每当用java命令启动一个Java虚拟机进程（ Application 应用程序），Java 虚拟机就会创建一个主线程，该线程**从程序入口main()方法开始执行。**
- 当在主线程中创建 Thread 类或其子类对象时，就创建了一个线程对象。主线程就是**上述创建线程的父线程。**
- Programmer可以控制线程的启动、挂起与终止。

线程的状态

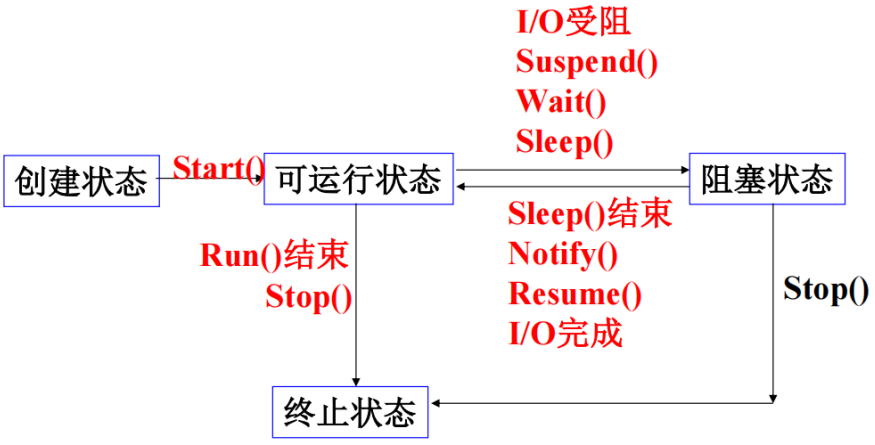
新建、就绪、运行、阻塞、终止



- 新建：当一个 Thread 类或其子类对象**被创建时**，新产生的线程处于新建状态，此时它已经有了**相应的内存空间和其他资源。**
- 就绪：调用 **start( )** 方法来启动处于新建状态的线程后，将进入线程队列排队等待 CPU 服务，此时它已经具备了运行的条件，**一旦轮到它来享用 CPU 资源时，就可以脱离创建**

它的主线程，开始自己的生命周期。

- 运行：当就绪状态的线程被调度并获得处理器资源时，便进入运行状态。
  - 每一个 Thread 类及其子类的对象都有一个重要的 run( ) 方法，当线程对象被调用执行时，它将自动调用本对象的 run( )方法，从第一句开始顺序执行。
  - **run( )** 方法定义了这个线程的操作和功能。
- 阻塞：一个正在执行的线程暂停自己的执行而进入的状态。引起线程由运行状态进入阻塞状态的可能情况：
  - 该线程正在等待 I/O 操作的完成：**等待 I/O 操作完成或回到就绪状态**
  - 网络操作
  - 为了获取锁而进入阻塞操作
  - 调用了该线程的 sleep( ) 方法：等待其指定的休眠事件结束后，**自动脱离阻塞状态**，回到就绪状态
  - 调用了 wait( ) 方法：调用 **notify**( )或 **notifyAll**( ) 方法；
  - 让处于运行状态的线程调用另一个线程的join()方法
- 终止：
  + 自然终止：线程完成了自己的全部工作
  + 强制终止：在线程执行完之前，调用stop( ) 或 destroy( ) 方法终止线程



4 创建和启动线程

Java中实现多线程有三种方法：

- 一种是继承Thread类；
- 第二种是实现Runnable接口；
- 第三种是实现Callable接口；

Thread构造方法

- Thread的构造方法一共有八个，这里根据命名方式分类，使用默认命名的构造方法如下
  - Thread()
  - Thread(Runnable target)
  - Thread(ThreadGroup group,Runnable target)
- 命名线程的构造方法如下：
  - Thread(String name)
  - Thread(Runnable target,Strintg name)
  - Thread(ThreadGroup group,String name)
  - Thread(ThreadGroup group,Runnable target,String name)
  - Thread(ThreadGroup group,Runnable target,String name,long stackSize)

- 一个线程的创建肯定是由另一个线程完成的；
- 被创建线程的父线程是创建它的线程；
- main线程由JVM创建，而main线程又可以成为其他线程的父线程；
- 如果一个线程创建的时候没有指定ThreadGroup，那么将会和父线程同一个ThreadGroup。main线程所在的ThreadGroup称为main；

Thread常用方法

Thread的主要方法(共34个,其中5个不再使用了)			
static Thread currentThread();		int getPriority();	
String getName();		ThreadGroup getThreadGroup();	
void interrupt();		boolean isInterrupted();	
void join();	join(long millis);	join(long millis, int nanos);	
static void yield();		void setName(String name);	
void setPriority(int priority);		void start();	
boolean isAlive();	static void sleep(long millis, int nanos);		
void run();	static void sleep(long millis);		

1. currentThread()：返回当前运行的Thread对象。
2. setName()：设置线程的名字。
3. getName()：返回该线程的名字。
4. setPriority()：设置线程优先级。
5. getPriority()：返回线程优先级。
6. start()：启动一个线程。
7. run()：线程体,由start()方法调用,当run()方法返回时,当前的线程结束。
8. stop()：使调用它的线程立即停止执行。
9. isAlive()：如果线程已被启动并且未被终止，那么isAlive()：返回true。如果返回false，则该线程是新创建或是已被终止的。
10. sleep(int n)：使线程睡眠n毫秒,n毫秒后,线程可以再次运行。
11. yield()：将CPU控制权主动移交到下一个可运行线程。
12. join()：方法join()将引起现行线程等待，直至方法join所调用的线程结束。
13. suspend()：使线程挂起,暂停运行。
14. resume() 恢复挂起的线程,使其处于可运行状态（Runnable）。
15. wait()：
16. notify()：
17. notifyAll()：

suspend();
resume();
Stop();
目前已经不再使用。为什么？请看API.

tips

因为Java线程的调度不是分时的，所以你必须确保你的代码中的线程会**不时地给另外一个线程运行的机会**。有三种方法可以做到一点：

- 让处于运行状态的线程调用 **Thread.sleep()** 方法。

- 让处于运行状态的线程调用 **Thread.yield()** 方法。
- 让处于运行状态的线程调用另一个线程的 **join()** 方法。

sleep与yield

- 这两个方法都是静态的实例方法。
- sleep()会有**中断异常抛出**，而yiled()不抛出任何异常。
- sleep()方法具有更好的**可移植性**，因为yield()的实现还取决于底层的操作系统对线程的调度策略。
- 对于yield()的主要用途是在**测试阶段，人为的提高程序的并发性能，以帮助发现一些隐藏的并发错误**，当程序正常运行时，则不能依靠yield方法提高程序的并发行能。

wait与sleep

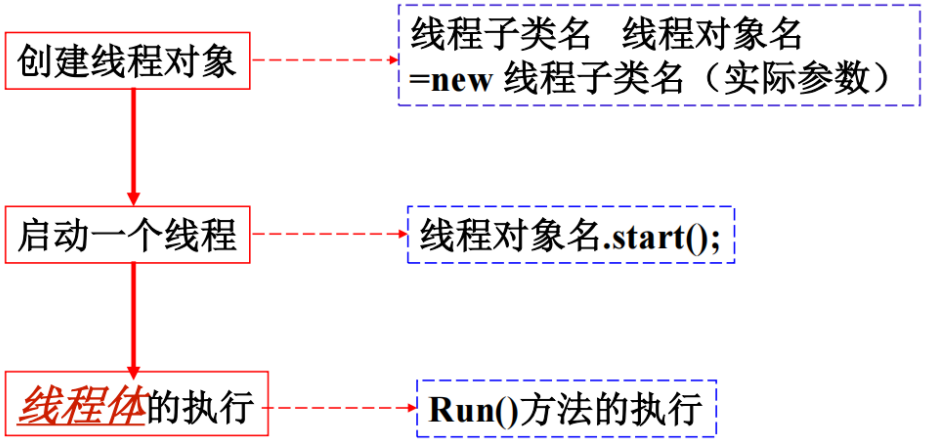
- 所以，wait,notify和notifyAll都是与同步相关联的方法,只有在synchronized方法中才可以用。在不同步的方法或代码中则使用sleep()方法使线程暂时停止运行

join

作用：使当前正在运行的线程暂停下来，**等待指定的时间后**或**等待调用该方法的线程结束后**，再恢复运行

应用线程类Thread创建线程

- 将一个类定义为Thread的子类,那么这个类就可以用来创建线程。
- 这个类中有一个至关重要的方法——`public void run`，这个方法称为**线程体**，它是整个线程的**核心**，线程所要完成任务的代码都定义在**线程体**中，实际上**不同功能的线程之间的区别就在于它们线程体的不同**



应用Runnable接口创建线程

- Runnable是Java中用以实现线程的接口，从根本上讲，任何实现线程功能的类都必须实现该接口。
  - Thread(Runnable target);
  - Thread(Runnable target, String name);
- Runnable接口中只定义了一个方法就是run()方法，也就是线程体

适用于采用实现Runnable接口方法的情况

- 避免**单继承**的局限：因为Java只允许单继承，如果一个类已经继承了Thread，就不能再继承其他类。
- 特别是在除了**run()方法**以外,并不打算重写Thread类的其它方法的情况下,以实现Runnable接口的方式生成新线程就显得更加合理了。
- 涉及到数据共享的时候；

终止线程

- 当线程执行完run()方法，它将自然终止运行。
- Thread有一个stop()方法，可以强制结束线程，但这种方法是不安全的。因此，在stop()方法已经被废弃。
- 实际编程中，一般是定义一个标志变量，然后通过程序来改变标志变量的值，从而控制线程从run()方法中自然退出

总结：创建用户多线程的步骤

法1

1.创建一个Thread类的子类
2.在子类中将希望该线程做的工作写到run()里面
3.生成该子类的一个对象
4.调用该对象的start()方法

```
class MyThread extends Thread{
    public void run(){... ...}
    //其它方法等
}
class MyClass{
    public static void main(String[] args){
        MyThread mt = new MyThread();
        mt.start();
    }
    //其它方法等
}
```

省略号代表的是我们想让这个线程完成的工作

调用start(),就会生成一个新的线程,并开始执行run()里规定的任务

2024/12/16

法2

法3

- 在程序开发中只要是多线程**尽量以实现Runnable接口为主**，因为实现Runnable接口相比继承Thread类有如下好处：
  - 避免单继承的局限，一个类可以实现多个接口。
  - 适合于资源的共享
- Runnable的局限性
  - `run()` 方法的返回值是void
  - 不允许抛出任何已检查的异常（编译时捕获的异常）

Callable接口

实现callable接口的步骤

1. **创建一个实现Callable接口的类。**
2. 重写类中的**Callable接口的call()方法。**
3. **通过传递实现了Callable接口的类对象来创建FutureTask对象。**
4. **通过传递创建的FutureTask对象在主类中创建一个Thread对象。**
5. 调用**Thread对象的start()方法。**
6. **使用FutureTask对象的get()方法获取可调用类返回的结果。**

```
CallableThread callableThread = new CallableThread();
FutureTask<String> futureTask = new FutureTask<>(callableThread);
Thread thread = new Thread(futureTask);
thread.start();
String result = futureTask.get();
System.out.println("执行结果= " + result);
```

使用 FutureTask.get()方法获取执行结果

借助FutureTask，因为 Thread类没有接受Callable的构造函数。

callable接口的特点

- **Callable接口的 call() 方法可以返回任何对象。**
- **Callable接口的 call() 方法可以抛出已检查的异常（编译时捕获的异常。例如：Exception）和未检查的异常（运行时发生的异常。例如：RuntimeException）。**
- **可以使用FutureTask对象的cancel()方法取消计算。**

- 运行Callable任务可以拿到一个Future对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过Future对象**可以了解任务执行情况**，可取消任务的执行，还可获取执行结果。

线程池

日常开发中，推荐使用**线程池**的方式来使用。最开始创建一堆线程放在池子里，用的时候拿出来用，不用就放回去，能够减少线程的启动和灭亡

实际工作中如何选择

- 取舍的基本原则就是需不需要**返回值**，如果不需要返回值，那直接就选 Runnable。如果有返回值的话，使用Callable。
- 另外一点就是是否需要**抛出异常**， Runnable是不接受抛出异常的，Callable可以抛出异常。
- Runnable适合那种纯异步的处理逻辑。比如每天定时计算报表，将报表存储到数据库或者其他地方，只是要计算，**不需要马上展示**，展示内容是在其他的方法中单独获取的。（比如那些非核心的功能，当核心流程执行完毕后，非核心功能就自己去执行）
- Callable适用于那些需要返回值或者需要抛出checked exception的情况，比如对某个任务的计算结果进行处理。在Java中，常常使用callable来实现异步任务的处理，以提高系统的吞吐量和响应速度

网络编程 14

网络编程

- C/S，B/S
- URI，URL，TCP，UDP
- Socket
- ...

了解基本概念即可，不会考太难。

计算机网络工作模式

- 客户机/服务器模式(Client/Server C/S)
  一共两种
  - **数据库服务器端**，客户端**通过数据库连接访问服务器端的数据**；
  - （本讲内容）**Socket服务器端**，服务器端的程序通过Socket与客户端的程序通信。另，socket服务器端为“**传输层**”，BS模式为“应用层”
- 浏览器/服务器模式（Browser/Server）

网络通信协议与接口

- 网络通信协议：计算机网络中实现通信必须有一些约定
- 网络通信接口：为了使两个结点之间能进行对话，必须在他们之间建立通信工具(即**接口**)，使彼此之间能进行信息交换，接口包括两部分：
  + 硬件装置:实现结点之间的信息传递。

+ 软件装置:规定双方进行通信的约定协议。

URI 包含 URL 和 URN

URI（Uniform Resource Identifier，统一资源**标识**符）用于唯一地**标识**资源，无论是通过名称、位置还是两者兼有。
URL（Uniform Resource Locator，统一资源**定位**符）是URI的一个子集，它提供了资源的**定位**信息，即如何访问资源，但不直接提供资源的名称。
URN（Uniform Resource Name）是URI的另一个子集，它提供了资源的**名称**，但不提供如何定位或访问资源的信息。URN是持久的、与位置无关的标识符。

TCP/IP

- TCP/IP是一组在Internet网络上的不同计算机之间进行通信的协议的总称；从下往上可视为4层结构：*物理层、网络层、传输层和应用层。*
- TCP/IP由应用层的HTTP、FTP、SMTP和传输层的TCP（传输控制协议）、UDP（用户数据报协议）以及网络层的IP（Internet协议）等一系列协议组成。

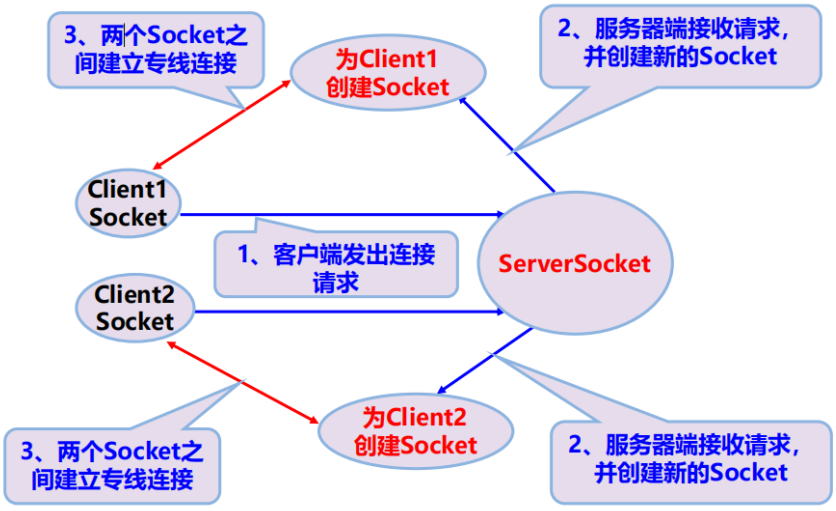
TCP和UDP

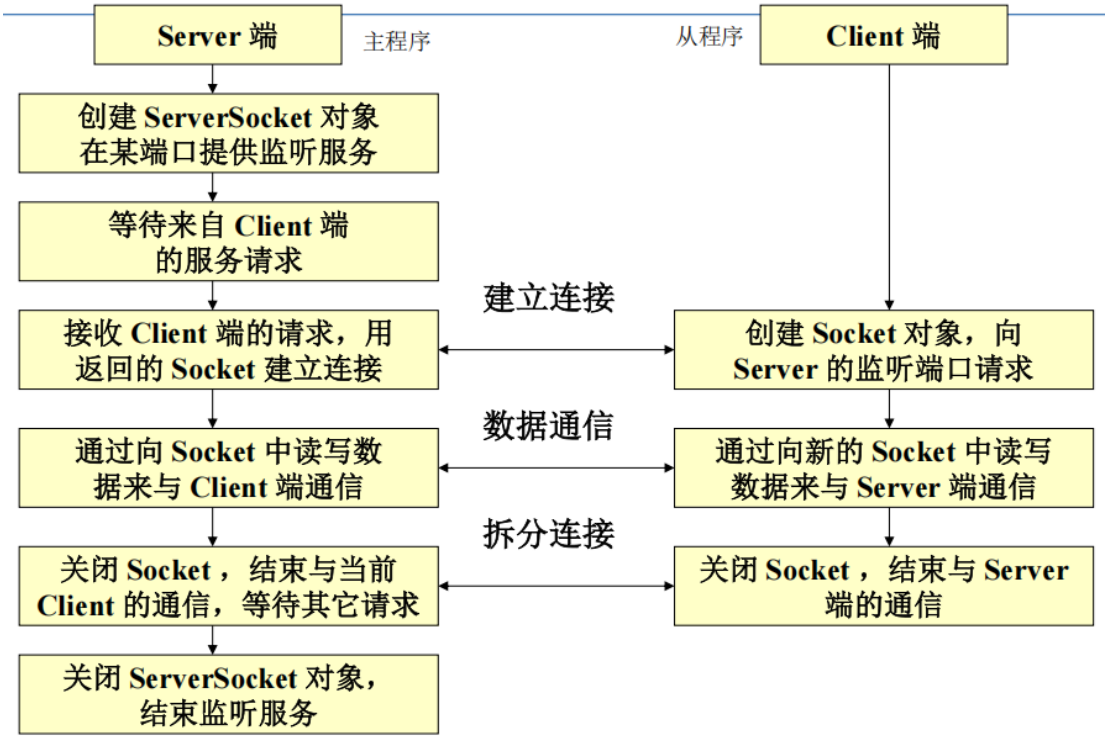
- UDP与TCP
  - TCP，传输控制协议(Transmission Control Protocol)，是面向连接的通信协议。
    - 使用TCP协议进行数据传输时,两个进程之间会建立一个连接,数据以流的形式顺序传输。
  - UDP，用户数据协议(User Datagram Protocol)，是无连接通信协议。
    - 使用UDP协议进行数据传输时,两个进程之间不建立特定的连接,不对数据到达的顺序进行检查。
- 在互联网上进行数据传输,多用TCP和UDP协议，它们传输的都是一个byte stream/字节型的数据流

更加稳定可靠，常用来传文件

效率高但容易丢东西

TCP网络程序的工作原理





在TCP网络连接上传递对象

- ObjectInputStream和ObjectOutputStream可以用来包装底层网络字节流，TCP服务器和TCP客户端之间就可以传递对象类型的数据，实现从底层输入流中读取对象类型的数据和将对象类型的数据写入到底层输出流。
- RMI(remote method invocation)编程：是java进行分布式编程的基础。

UDP网络程序

- 用户数据报协议UDP（user datagram protocol）是一个无连接的、发送独立数据包的协议，它不保证数据按顺序传送和正确到达。
- 数据报Socket又称为UDP套接字，它无需建立、拆除连接，而是直接将信息打包传向指定的目的地，使用简单，占用资源少，适合于断续、非实时通信。
- 利用UDP通信的两个程序是平等的，没有主次之分，两个程序的代码可以完全一样。

设计原则(7个) 7

SOLID合成复用

1. S **Single Responsibility Principle (单一职责原则)**：每个类只干一件事
2. O **Open/Closed Principle (开闭原则)**：用抽象类和接口而不是if-else
3. L **Liskov Substitution Principle (里氏代换原则)**：子类不能改变父类的方法
4. I **Interface Segregation Principle (接口隔离原则)**：把总接口拆分成多个接口（防止有的类不需要某个功能但被迫实现）
5. D **Dependency Inversion Principle (依赖倒转原则)**：细节(更具体的东西，如email通信)实现抽象(更抽象的东西，如通信)而不是抽象拥有细节
6. **迪米特法则**：一个软件实体尽量少的与其他实体发生相互作用（找中介）
7. **合成复用原则**：少用继承，用组合/聚合代替继承

设计原则

- **单一职责**原则

一个类只负责一个功能领域中的相应职责。

eg. 有一个图形绘制类，它既负责绘制圆形又负责绘制矩形和三角形等多种图形，这样当需要修改圆形的绘制算法时，可能会影响到其他图形的绘制。按照单一职责原则，可以将其拆分成圆形绘制类、矩形绘制类和三角形绘制类等，每个类只负责一种图形的绘制，这样修改其中一个类时不会影响到其他类

- **开闭原则**

一个软件实体应当对扩展开放，对修改关闭。

```
class GraphicDrawer {
    public void drawShape(Shape shape) {
        if (shape instanceof Circle) {
            // 绘制圆形的代码
        } else if (shape instanceof Rectangle) {
            // 绘制矩形的代码
        }
    }
}
```

```
class Circle implements Shape {
    // 圆形的属性和方法
}

class Rectangle implements Shape {
    // 矩形的属性和方法
}
```

- **合成复用**原则

尽量使用组合或者聚合关系实现代码复用，少使用继承。

继承的耦合度很高

- **依赖倒转**原则

抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。

```
class Computer {
    private CPU cpu;
    private Memory memory;

    public Computer() {
        this.cpu = new IntelCPU();
        this.memory = new KingstonMemory();
    }
}

class IntelCPU {
    // CPU
}

class KingstonMemory {
    // 内存
}
```

```
interface CPU {
    void process();
}

interface Memory {
    void storeData();
}

class Computer {
    private CPU cpu;
    private Memory memory;

    public Computer(CPU cpu, Memory memory) {
        this.cpu = cpu;
        this.memory = memory;
    }
}
```

- **接口隔离**原则

使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。

- **迪米特法则**

一个软件实体应当尽可能少地与其他实体发生相互作用。

- **里氏代换**原则

核心思想是：子类对象应该能够替换其父类对象，而不会导致程序的业务逻辑出现异常。即子类应该继承父类的所有属性和行为，并且可以在此基础上进行扩展，但不能改变父类原有的行为。

所有引用基类（父类）的地方必须能透明地使用其子类的对象

开闭

在Java中，开闭原则可以通过抽象类和接口来实现，这样可以在不修改现有代码的情况下扩展功能。以下是一个简单的Java例子，展示了如何遵循开闭原则。

不符合开闭原则的例子：

```
public class Rectangle {
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
public class Circle {
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
public class GraphicEditor {
    public void drawShape(Object shape) {
        if (shape instanceof Rectangle) {
            ((Rectangle) shape).draw();
        } else if (shape instanceof Circle) {
            ((Circle) shape).draw();
        }
        // 如果要添加新的图形，比如三角形，需要修改这个方法
    }
}
public class Main {
    public static void main(String[] args) {
        GraphicEditor editor = new GraphicEditor();
        editor.drawShape(new Rectangle());
        editor.drawShape(new Circle());
    }
}
```

在这个例子中，`GraphicEditor` 类的 `drawShape` 方法依赖于具体的图形类。如果我们要添加一个新的图形类，比如三角形，我们需要修改 `GraphicEditor` 类，这违反了开闭原则。

符合开闭原则的例子：

```
public interface Shape {
    void draw();
}
public class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
```

```
public class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
public class GraphicEditor {
    public void drawShape(Shape shape) {
        shape.draw();
    }
}
public class Triangle implements Shape {
    public void draw() {
        System.out.println("Drawing a triangle");
    }
}
public class Main {
    public static void main(String[] args) {
        GraphicEditor editor = new GraphicEditor();
        editor.drawShape(new Rectangle());
        editor.drawShape(new Circle());
        // 添加新的图形时，不需要修改GraphicEditor类
        editor.drawShape(new Triangle());
    }
}
```

在这个改进的例子中，我们定义了一个 `Shape` 接口，所有的图形类都实现这个接口。`GraphicEditor` 类的 `drawShape` 方法现在接受一个 `Shape` 接口类型的参数，而不是具体的图形类。这样，当我们需要添加新的图形类（如 `Triangle`）时，我们只需要创建一个新的类实现 `Shape` 接口，而不需要修改 `GraphicEditor` 类。这符合开闭原则，因为我们对扩展是开放的，对修改是关闭的。

合成复用1

当然可以。下面是一个更简单的例子，用于说明合成复用原则：
场景：我们有一个表示汽车的类，汽车可以有不同的引擎。而不是通过继承来创建不同类型的汽车，我们使用组合来复用引擎的行为。
不使用合成复用原则的例子（使用继承）：

```
class Engine {
    public void start() {
        System.out.println("Engine starts");
    }
}
class CarWithPetrolEngine extends Engine {
    // 使用汽油引擎的汽车
}
```

```
class CarWithDieselEngine extends Engine {
    // 使用柴油引擎的汽车
}
public class Main {
    public static void main(String[] args) {
        CarWithPetrolEngine petrolCar = new CarWithPetrolEngine();
        petrolCar.start(); // Engine starts
        CarWithDieselEngine dieselCar = new CarWithDieselEngine();
        dieselCar.start(); // Engine starts
    }
}
```

在这个例子中，我们通过继承来创建不同类型的汽车，但这可能导致不必要的复杂性，尤其是当引擎类型增多时。

使用合成复用原则的例子（使用组合）：

```
interface Engine {
    void start();
}
class PetrolEngine implements Engine {
    public void start() {
        System.out.println("Petrol engine starts");
    }
}
class DieselEngine implements Engine {
    public void start() {
        System.out.println("Diesel engine starts");
    }
}
class Car {
    private Engine engine;
    public Car(Engine engine) {
        this.engine = engine;
    }
    public void start() {
        engine.start();
    }
}
public class Main {
    public static void main(String[] args) {
        Car petrolCar = new Car(new PetrolEngine());
        petrolCar.start(); // Petrol engine starts
        Car dieselCar = new Car(new DieselEngine());
        dieselCar.start(); // Diesel engine starts
    }
}
```

在这个改进的例子中，我们定义了一个 `Engine` 接口和两个实现类 `PetrolEngine` 和 `DieselEngine`。`Car` 类有一个 `Engine` 类型的成员变量，并在构造函数中注入具体的引擎。这样，我们可以通过组合不同的引擎来创建不同类型的汽车，而不是通过继承。
这种方法的好处是，如果将来我们有新的引擎类型（例如电动引擎），我们只需要添加一个新的实现类，而不需要修改 `Car` 类或其子类。这提高了代码的复用性、灵活性和可维护性。

合成复用2

合成复用原则（Composite Reuse Principle）是面向对象设计的原则之一，它建议在设计中要尽量使用对象组合，而不是继承来达到复用的目的。该原则强调通过组合不同的对象来获得新的功能，而不是通过继承来扩展类的功能。这样可以减少系统的复杂性，提高灵活性和可维护性。
以下是一个Java例子，展示了如何应用合成复用原则：
不使用合成复用原则的例子（使用继承）：

```
class Bird {
    public void fly() {
        System.out.println("Bird is flying");
    }
}
class Sparrow extends Bird {
    // Sparrow继承了Bird的fly方法
}
class Penguin extends Bird {
    // Penguin继承了Bird的fly方法，但企鹅不会飞，这导致了不合理的设计
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Penguin cannot fly");
    }
}
public class Main {
    public static void main(String[] args) {
        Sparrow sparrow = new Sparrow();
        sparrow.fly(); // 合理
        Penguin penguin = new Penguin();
        penguin.fly(); // 不合理，抛出异常
    }
}
```

在这个例子中，`Penguin` 类继承了 `Bird` 类，但企鹅不会飞，所以继承导致了不合理的设计。我们需要重写 `fly` 方法来抛出异常，这违反了合成复用原则。
使用合成复用原则的例子（使用组合）：

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

```
class Bird {
    private Flyable flyable;
    public Bird(Flyable flyable) {
        this.flyable = flyable;
    }
    public void performFly() {
        flyable.fly();
    }
}
class Sparrow extends Bird {
    public Sparrow() {
        super(new Flyable() {
            public void fly() {
                System.out.println("Sparrow is flying");
            }
        });
    }
}
class Penguin extends Bird {
    public Penguin() {
        super(new Flyable() {
            public void fly() {
                System.out.println("Penguin cannot fly");
            }
        });
    }
}
class FlyWithWings implements Flyable {
    public void fly() {
        System.out.println("Flying with wings");
    }
}
class NoFly implements Flyable {
    public void fly() {
        System.out.println("Cannot fly");
    }
}
public class Main {
    public static void main(String[] args) {
        Sparrow sparrow = new Sparrow();
        sparrow.performFly(); // 合理
        Penguin penguin = new Penguin();
        penguin.performFly(); // 合理，输出"Penguin cannot fly"
    }
}
```

在这个改进的例子中，我们定义了一个 `Flyable` 接口，表示飞行的能力。`Bird` 类有一个 `Flyable` 类型的成员变量，并在构造函数中注入飞行行为。`Sparrow` 和 `Penguin` 类通过

构造函数分别注入了不同的飞行行为。这样，我们通过组合而不是继承来实现了复用，符合合成复用原则。
通过这种方式，我们可以更容易地添加新的飞行行为或者修改现有的行为，而无需修改 `Bird` 类或其子类的代码，从而提高了系统的灵活性和可维护性。

依赖倒转

依赖倒转原则（Dependency Inversion Principle，DIP）是面向对象设计原则之一，也是SOLID原则中的“D”。它主张：

1. 高层模块不应该依赖低层模块，两者都应该依赖抽象。
2. 抽象不应该依赖细节，细节应该依赖抽象。
   简单例子：假设我们有一个通知系统，最初只通过邮件发送通知。
   违反依赖倒转原则的代码：

```
class EmailService {
    public void sendEmail(String message, String recipient) {
        // 发送邮件的实现
    }
}
class NotificationService {
    private EmailService emailService;
    public NotificationService() {
        this.emailService = new EmailService();
    }
    public void notify(String message, String recipient) {
        emailService.sendEmail(message, recipient);
    }
}
```

在这个例子中，`NotificationService` 直接依赖于 `EmailService` 的具体实现，这违反了依赖倒转原则。
遵循依赖倒转原则的改进代码：

```
interface NotificationService {
    void notify(String message, String recipient);
}
class EmailService implements NotificationService {
    public void notify(String message, String recipient) {
        // 发送邮件的实现
    }
}
class SMSService implements NotificationService {
    public void notify(String message, String recipient) {
        // 发送短信的实现
    }
```

```
}
class NotificationController {
    private NotificationService notificationService;
    public NotificationController(NotificationService
notificationService) {
        this.notificationService = notificationService;
    }
    public void sendNotification(String message, String recipient) {
        notificationService.notify(message, recipient);
    }
}
```

在这个改进的例子中，我们引入了 `NotificationService` 接口，作为抽象。
`EmailService` 和 `SMSService` 都实现了这个接口。 `NotificationController` 类依赖于 `NotificationService` 接口，而不是具体的实现。
这样，如果将来我们需要添加新的通知方式（如微信、推送通知等），我们只需要创建新的实现类即可，而不需要修改 `NotificationController`。这提高了代码的灵活性和可维护性。
总结：依赖倒转原则通过依赖抽象而不是具体实现，降低了模块间的耦合度，使得系统更易于扩展和维护。

接口隔离原则

假设我们有一个用于打印文档的接口 `Printer`，它包含以下方法：

- `printDocument()`
- `scanDocument()`
- `faxDocument()`
- `printPhoto()`

现在，我们有几个不同的类实现了这个接口：

1. **SimplePrinter**：一个基本的打印机，只能打印文档。
2. **MultiFunctionPrinter**：一个多功能打印机，可以打印、扫描、传真和打印照片。

根据接口隔离原则， `SimplePrinter` 类不应该被迫实现 `scanDocument()`、`faxDocument()` 和 `printPhoto()` 这些它不需要的方法。这样做会导致 `SimplePrinter` 类包含冗余的、不相关的代码。

应用接口隔离原则后的改进：
我们可以将 `Printer` 接口拆分成更小的接口：

- `PrintDocumentInterface`：包含 `printDocument()` 方法。
- `ScanDocumentInterface`：包含 `scanDocument()` 方法。
- `FaxDocumentInterface`：包含 `faxDocument()` 方法。
- `PrintPhotoInterface`：包含 `printPhoto()` 方法。

然后，我们的类可以按需实现这些接口：

1. **SimplePrinter**：实现 `PrintDocumentInterface`。
2. **MultiFunctionPrinter**：实现 `PrintDocumentInterface`、`ScanDocumentInterface`、`FaxDocumentInterface` 和 `PrintPhotoInterface`。

迪米特法则

在Java中，迪米特法则（Law of Demeter）同样强调减少类之间的直接交互，以降低耦合度。以下是一个简单的Java例子，用于说明如何应用迪米特法则。
场景： 假设我们有一个订单处理系统，其中包含 `Order`（订单）、`Customer`（客户）和 `Payment`（支付）等类。
不符合迪米特法则的设计：

```
class Order {
    private Customer customer;
    private Payment payment;
    public void process() {
        // Order类直接与Customer和Payment类交互
        customer.verify();
        payment.pay();
    }
}
class Customer {
    public void verify() {
        // 客户验证逻辑
    }
}
class Payment {
    public void pay() {
        // 支付逻辑
    }
}
```

在这个例子中，`Order` 类直接调用了 `Customer` 和 `Payment` 类的方法，这意味着 `Order` 类需要了解 `Customer` 和 `Payment` 类的内部实现。这违反了迪米特法则。
符合迪米特法则的设计：
为了遵守迪米特法则，我们可以引入一个中介者类，例如`OrderProcessor`，来处理订单处理的逻辑：
我们还可以进一步封装 `Order` 类，使其不直接暴露 `Customer` 和 `Payment` 对象。通过这种方式，我们进一步限制了 `Order` 类与其他类的直接交互，使得类之间的关系更加清晰，符合迪米特法则。

```
class Order {
    private Customer customer;
    private Payment payment;
    public void process(OrderProcessor processor) {
```

```
            processor.process(this);
        }
        // 私有方法，用于OrderProcessor访问
        private Customer getCustomer() {
            return customer;
        }
        // 私有方法，用于OrderProcessor访问
        private Payment getPayment() {
            return payment;
        }
    }
    class Customer {
        public void verify() {
            // 客户验证逻辑
        }
    }
    class Payment {
        public void pay() {
            // 支付逻辑
        }
    }
    class OrderProcessor {
        public void process(Order order) {
            // OrderProcessor负责处理订单，与Customer和Payment类交互
            order.getCustomer().verify();
            order.getPayment().pay();
        }
    }
```

在这个改进后的设计中，`Order` 类不再直接与 `Customer` 和 `Payment` 类交互，而是通过 `OrderProcessor` 类来进行。这样，`Order` 类不需要了解 `Customer` 和 `Payment` 类的内部实现，从而减少了类之间的耦合。
迪米特法则的应用有助于创建松耦合、高内聚的类设计，从而提高代码的可维护性和可扩展性。然而，也需要注意不要过度应用，以免导致代码过于复杂或难以理解。

里氏代换原则

里氏代换原则（Liskov Substitution Principle，LSP）是面向对象设计中的五大原则之一，由芭芭拉·利斯科夫（Barbara Liskov）在1987年提出。该原则的核心思想是：子类对象应该能够替换其父类对象，而不会导致程序的业务逻辑出现异常。
里氏代换原则强调的是子类和父类之间的兼容性，即子类应该继承父类的所有属性和行为，并且可以在此基础上进行扩展，但不能改变父类原有的行为。这样，在程序中，我们可以放心地使用父类对象的地方替换为子类对象，而不会影响程序的正确性。
以下是一个Java例子，用于说明里氏代换原则：
场景： 假设我们有一个表示鸟的基类`Bird`，以及两个子类 `Sparrow`（麻雀）和`Ostrich`

（鸵鸟）。

不符合里氏代换原则的设计：

```
class Bird {
    public void fly() {
        System.out.println("This bird can fly.");
    }
}
class Sparrow extends Bird {
    // Sparrow继承了Bird的fly方法，可以飞行
}
class Ostrich extends Bird {
    // Ostrich也继承了Bird的fly方法，但鸵鸟实际上不会飞
    // 这违反了里氏代换原则，因为Ostrich不能替换Bird而不改变程序的预期行为
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Ostrich cannot
fly.");
    }
}
public class Main {
    public static void main(String[] args) {
        Bird bird = new Sparrow();
        bird.fly(); // 正常运行
        bird = new Ostrich();
        bird.fly(); // 这里会抛出异常，违反了里氏代换原则
    }
}
```

在这个例子中，`Ostrich` 类继承了 `Bird` 类的 `fly` 方法，但实际上鸵鸟是不会飞的。当我们尝试用 `Ostrich` 对象替换 `Bird` 对象时，调用 `fly` 方法会抛出异常，这违反了里氏代换原则。

符合里氏代换原则的设计：

为了遵守里氏代换原则，我们不应该让 `Ostrich` 继承 `Bird` 的 `fly` 方法。我们可以通过提取接口或使用组合的方式来解决这个问题：

```
interface Flyable {
    void fly();
}
class Bird {
    // Bird类不再包含fly方法
}
class Sparrow extends Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Sparrow can fly.");
```

```
    }
}
class Ostrich extends Bird {
    // Ostrich不再继承fly方法，因此不会违反里氏代换原则
}
public class Main {
    public static void main(String[] args) {
        Flyable flyableBird = new Sparrow();
        flyableBird.fly(); // 正常运行
        Bird bird = new Ostrich();
        // 鸵鸟没有fly方法，但我们也没有期望它能够飞行
        // 这符合里氏代换原则，因为Ostrich可以替换Bird而不改变程序的预期行为
    }
}
```

在这个改进后的设计中，我们引入了 `Flyable` 接口，只有会飞的鸟（如 `Sparrow`）才实现这个接口。`Ostrich` 类不再继承 `fly` 方法，因此不会违反里氏代换原则。这样，我们可以确保在程序中替换父类对象为子类对象时，不会影响程序的正确性。
里氏代换原则是面向对象设计中的重要原则，它有助于我们设计出更加灵活、可扩展和可维护的代码。通过遵守这个原则，我们可以确保子类和父类之间的兼容性，避免在程序运行时出现意外行为。

设计模式 7

- 创建型

  单例模式，工厂模式（简单工厂、抽象工厂），原型模式

- 结构型

  适配器模式，装饰模式，门面模式

- 行为型

  策略模式，访问者模式，责任链模式，观察者模式

工厂方法模式的核心是把类的**实例化延迟到其子类**
被造的东西有个接口、工厂有个接口，被造的东西和工厂分别实现这两个接口，然后工厂类 `public Vehicle createVehicle() { return new Car();}`
适配器模式的核心是将一个类的接口**转换**成客户希望的另外一个接口
两个接口：原来的东西和新加的东西。类来实现新加的东西的接口。Adapter实现原来的东西的接口，同时拥有新加的东西的接口。
装饰模式的核心是**动态**地给对象添加一些额外的职责。
具体组件继承抽象组件；抽象装饰继承抽象组件，同时拥有抽象组件；具体装饰继承抽象装饰，有装饰函数；调用时 `Bird bird = new Sparrow; bird = new birdDecorator(bird)`
外观模式的核心是通过为多个复杂的子系统提供一个**一致的接口**，而使这些子系统**更加容易被访问**的模式。

外观角色拥有子系统123；客户角色依赖外观角色。

策略模式的核心是定义**一系列算法**,把它们一个个**封装**起来,并且使它们可**相互替换**。本模式使得算法可独立于使用它的客户而变化。

上下文拥有抽象策略，同时在方法体内调用策略的算法；具体策略实现抽象策略

访问者模式的核心是在**不改变各个元素的类**的前提下定义作用于这些元素的**新操作**。

元素、访问者接口；具体元素实现元素接口；具体访问者实现访问者接口；具体元素和集体访问者相互关联

责任链模式的关键是将用户的请求**分派给许多对象**。

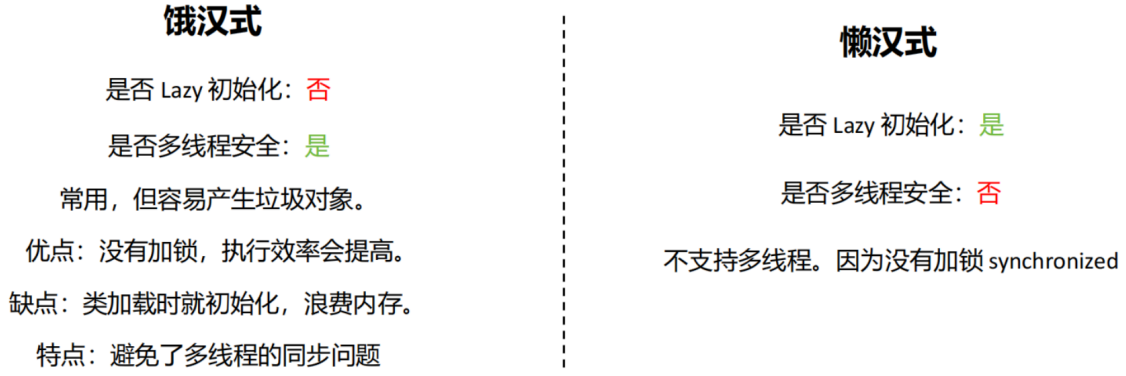
处理者接口规定具体处理者**处理用户的请求的方法**以及具体处理者**设置后继对象**的方法；具体处理者实现处理者接口；使用时先设置后继对象在调用第一个处理者

观察者模式的核心是当一个对象改变状态时，所有依赖于它的对象都会得到通知并自动更新。

被观察者存了一个list表示观察者，观察者存了自己观察的对象。当被观察者发生变化时，通知观察者，观察者更新数据并展示出来

单例模式

- **饿汉式**：类加载时就创建实例，像是一个饥饿的人急于吃东西。
- **懒汉式**：使用时才创建实例，像是一个懒惰的人等到需要时才行动。



6. 关于单例模式，以下说法正确的是：_____
A 它拥有公有的构造方法
B 它拥有公有实例方法getInstance()，以获得唯一的一个实例 是静态的 什么是实例？？
C 懒汉式单例，即单例实例在第一次被使用时构建；而饿汉式单例，即JVM加载这个类时就创建单例实例
D 其它选项均不正确
正确答案: C

饿汉式

```
public class Singleton {
    // 静态的。保留自身的引用，类加载时就初始化
    private static Singleton test = new Singleton();
    // 必须是私有的构造函数
    private Singleton() {}
    // 公共的静态方法
    public static Singleton getInstance() {
        return test;
    }
}
```

懒汉式

```
public class Singleton {
    // 静态的。保留自身的引用
    private static Singleton test = null;
    // 必须是私有的构造函数
    private Singleton() {}
    // 公共的静态方法
    public static Singleton getInstance() {
        if(test == null) {
            test = new Singleton();
        }
        return test;
    }
}
```

多线程安全的懒汉式

```
// 修改上述代码避免多线程中的安全问题
public class Singleton {
    private static Singleton test = null;
    private Singleton() {}
    public static Singleton getInstance() {
        if(test == null) {
                synchronized(Singleton.class){
                        if(test == null){
                            test = new Singleton();
                        }
                }
        }
        return test;
    }
}
```

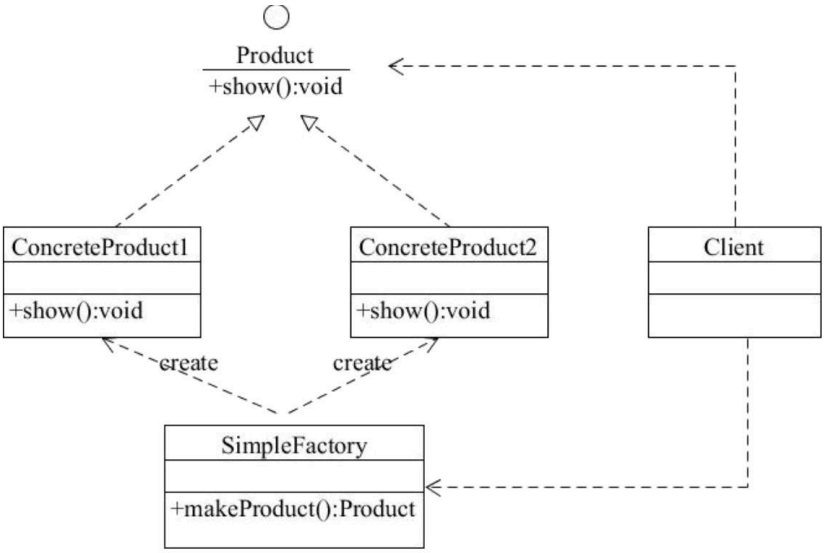
工厂模式

简单工厂：**一个工厂类**，由一**个工厂类根据传入的参数**决定创建哪一种产品类的实例。
工厂方法：**一个工厂接口，一堆工厂类（每种产品都有一个类），一条产品线（产出不同种类的东西）**。工厂方法模式定义了一个创建对象的接口，但**由子类决定要实例化的类是哪一个**。工厂方法模式让类的实例化推迟到子类。
抽象工厂：**一个工厂接口，一堆工厂类（每种产品都有一个类），一堆产品线（每条产品线能产出不同种类的东西）**，提供了一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。这种模式通常用于系统中有多个**产品族**，且每个产品族都有多个产品等级的情况。

工厂模式（Factory Pattern）是Java中最常用的设计模式之一。这种模式提供了一种创建对象的最佳方式，通过使用工厂模式，我们可以将对象的创建逻辑与使用逻辑分离，使得客户端代码不依赖于具体类的实现，而是依赖于抽象接口或类。这样，当需要更换或增加新的产品类时，不需要修改客户端代码，提高了代码的可扩展性和可维护性。
工厂模式主要有三种形式：

1. 简单工厂模式（Simple Factory Pattern）
2. 工厂方法模式（Factory Method Pattern）
3. 抽象工厂模式（Abstract Factory Pattern）
   下面以简单工厂模式和工厂方法模式为例，介绍工厂模式在Java中的实现。

简单工厂模式

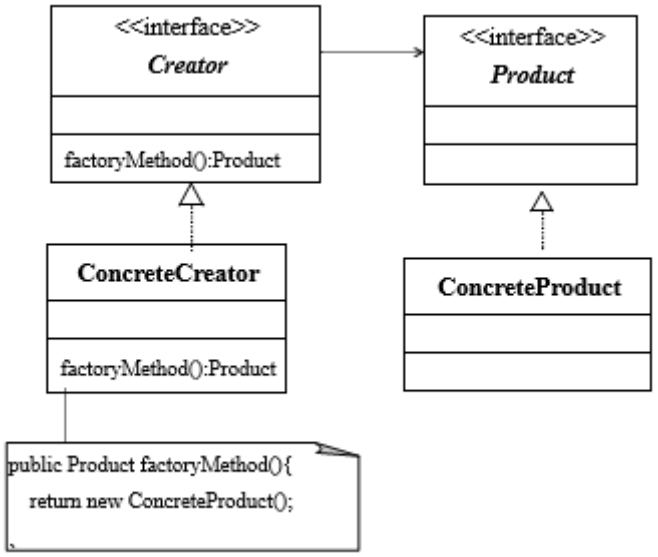


示例： 假设我们需要创建不同类型的交通工具，如汽车和自行车。

```
// 交通工具接口
interface Vehicle {
    void drive();
}
// 汽车类
class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car");
    }
}
// 自行车类
class Bicycle implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Riding a bicycle");
    }
}
// 简单工厂类
```

```
class VehicleFactory {
    public static Vehicle createVehicle(String type) {
        if ("car".equals(type)) {
            return new Car();
        } else if ("bicycle".equals(type)) {
            return new Bicycle();
        }
        throw new IllegalArgumentException("Unknown vehicle type");
    }
}
// 客户端代码
public class SimpleFactoryDemo {
    public static void main(String[] args) {
        Vehicle car = VehicleFactory.createVehicle("car");
        car.drive();
        Vehicle bicycle = VehicleFactory.createVehicle("bicycle");
        bicycle.drive();
    }
}
```

工厂方法模式



示例： 继续使用交通工具的例子，但这次我们将工厂类抽象化，并为每种交通工具提供一个具体的工厂类。

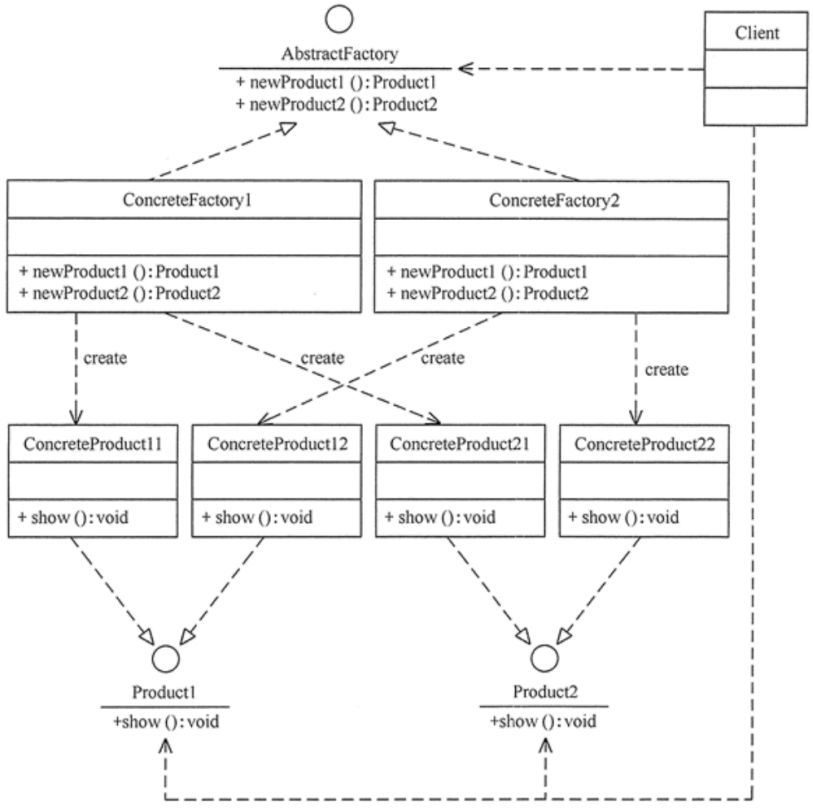
```
// 交通工具接口
interface Vehicle {
    void drive();
}
// 汽车类
class Car implements Vehicle {
    @Override
    public void drive() {
```

```
        System.out.println("Driving a car");
    }
}
// 自行车类
class Bicycle implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Riding a bicycle");
    }
}
// 工厂接口
interface VehicleFactory {
    Vehicle createVehicle();
}
// 汽车工厂类
class CarFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
}
// 自行车工厂类
class BicycleFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Bicycle();
    }
}
// 客户端代码
public class FactoryMethodDemo {
    public static void main(String[] args) {
        VehicleFactory carFactory = new CarFactory();
        Vehicle car = carFactory.createVehicle();
        car.drive();
        VehicleFactory bicycleFactory = new BicycleFactory();
        Vehicle bicycle = bicycleFactory.createVehicle();
        bicycle.drive();
    }
}
```

在这个例子中，我们定义了一个 `VehicleFactory` 接口和两个实现该接口的工厂类 `CarFactory` 和 `BicycleFactory`。每个工厂类负责创建一种类型的交通工具。客户端代码通过具体的工厂类来创建对象，这样当需要添加新的交通工具时，只需要添加新的工厂类和产品类，而不需要修改现有的代码。

工厂模式在Java中的应用非常广泛，它可以帮助我们更好地组织代码，实现解耦和灵活的对象创建。通过使用工厂模式，我们可以提高代码的可扩展性、可维护性和可测试性。

抽象工厂



要将上述代码改写成抽象工厂模式，我们需要定义一个抽象工厂接口，该接口不仅负责创建交通工具，还可能负责创建与交通工具相关的其他产品，比如轮胎（Tire）或引擎（Engine）。这样，每个具体工厂就能创建一个产品家族，而不仅仅是一个产品。
以下是一个简单的示例，展示如何将代码改写成抽象工厂模式：

```
// 交通工具接口
interface Vehicle {
    void drive();
}
// 汽车类
class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car");
    }
}
// 自行车类
class Bicycle implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Riding a bicycle");
    }
}
// 轮胎接口
interface Tire {
    void roll();
```

```
}
// 汽车轮胎类
class CarTire implements Tire {
    @Override
    public void roll() {
        System.out.println("Car tire rolling");
    }
}
// 自行车轮胎类
class BicycleTire implements Tire {
    @Override
    public void roll() {
        System.out.println("Bicycle tire rolling");
    }
}
// 抽象工厂接口
interface VehicleFactory {
    Vehicle createVehicle();
    Tire createTire();
}
// 汽车工厂类
class CarFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
    @Override
    public Tire createTire() {
        return new CarTire();
    }
}
// 自行车工厂类
class BicycleFactory implements VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Bicycle();
    }
    @Override
    public Tire createTire() {
        return new BicycleTire();
    }
}
// 客户端代码
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        VehicleFactory carFactory = new CarFactory();
        Vehicle car = carFactory.createVehicle();
        Tire carTire = carFactory.createTire();
```

```
        car.drive();
        carTire.roll();
        VehicleFactory bicycleFactory = new BicycleFactory();
        Vehicle bicycle = bicycleFactory.createVehicle();
        Tire bicycleTire = bicycleFactory.createTire();
        bicycle.drive();
        bicycleTire.roll();
    }
}
```

在这个改写后的例子中，`VehicleFactory` 接口现在有两个方法：`createVehicle` 和 `createTire`。每个具体工厂（`CarFactory` 和 `BicycleFactory`）都实现了这两个方法，分别用于创建交通工具和对应的轮胎。这样，每个工厂都能创建一个产品家族，而客户端代码可以通过抽象工厂接口来获取这些相关产品的实例。
抽象工厂模式的关键在于提供一个接口，用于创建多个相关或依赖对象的家族，而不需要明确指定具体类。这样，客户端代码就可以与具体类的实现细节解耦。

原型模式

原型模式（Prototype Pattern） 在Java中通常用于创建对象的一个副本，而不是通过构造函数重新创建。
主要角色：

1. **Prototype（原型接口）**：声明一个克隆自己的方法。
2. **ConcretePrototype（具体原型类）**：实现原型接口，实现克隆方法。
3. **Client（客户端）**：使用原型实例来创建新的对象。

示例：文档编辑器中的文档复制

假设我们有一个文档编辑器，用户可以创建文档，并希望能够复制现有的文档以创建新的文档。这里，文档对象就是一个原型。

1. 定义原型接口

```
public interface DocumentPrototype extends Cloneable {
    DocumentPrototype clone();
    void setContent(String content);
    String getContent();
}
```

2. 实现具体原型类

```
public class TextDocument implements DocumentPrototype {
    private String content;
    public TextDocument(String content) {
```

```
        this.content = content;
    }
    @Override
    public DocumentPrototype clone() {
        try {
            return (TextDocument) super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
    @Override
    public void setContent(String content) {
        this.content = content;
    }
    @Override
    public String getContent() {
        return content;
    }
}
```

3. 客户端代码使用原型

```
public class DocumentEditor {
    public static void main(String[] args) {
        // 创建一个初始文档
        TextDocument originalDocument = new TextDocument("Hello,
World!");
        System.out.println("Original Document Content: " +
originalDocument.getContent());
        // 复制文档
        TextDocument copiedDocument = (TextDocument)
originalDocument.clone();
        System.out.println("Copied Document Content: " +
copiedDocument.getContent());
        // 修改复制后的文档内容
        copiedDocument.setContent("Hello, Prototype Pattern!");
        System.out.println("Modified Copied Document Content: " +
copiedDocument.getContent());
        // 原始文档内容保持不变
        System.out.println("Original Document Content after copy
modification: " + originalDocument.getContent());
    }
}
```

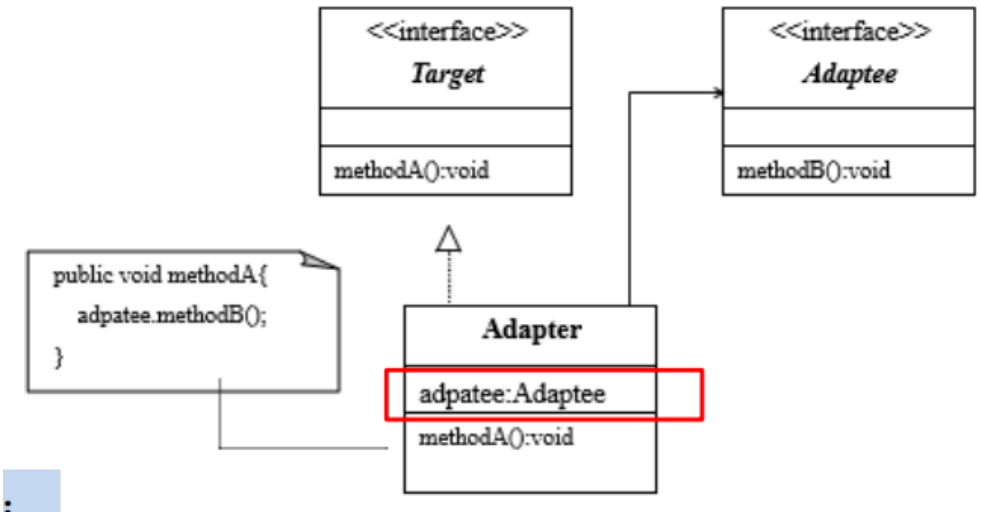
输出：

```
Original Document Content: Hello, World!
Copied Document Content: Hello, World!
Modified Copied Document Content: Hello, Prototype Pattern!
Original Document Content after copy modification: Hello, World!
```

说明：

- **DocumentPrototype** 接口定义了克隆方法，所有具体的文档类都需要实现这个接口。
- **TextDocument** 类实现了 **DocumentPrototype** 接口，并提供了具体的克隆实现。这里使用了Java的 `clone()` 方法，它执行的是深拷贝。
- 在 **DocumentEditor** 类中，我们创建了一个文档对象，并使用原型模式复制了这个对象。修改复制后的文档不会影响原始文档。
  这个例子展示了如何使用原型模式来复制对象，从而避免了通过构造函数重新创建对象的成本。

适配器模式



:

- **目标**（Target）：目标是一个接口，该接口是**客户想使用的接口。**
- **被适配者**（Adaptee）：被适配者是一个已经存在的**接口或抽象类**，这个接口或抽象类需要适配。
- **适配器**（Adapter）：适配器是一个类，该类实现了目标接口并包含有被适配者的引用，即**适配器的职责是对被适配者接口（抽象类）与目标接口进行适配。**

例1

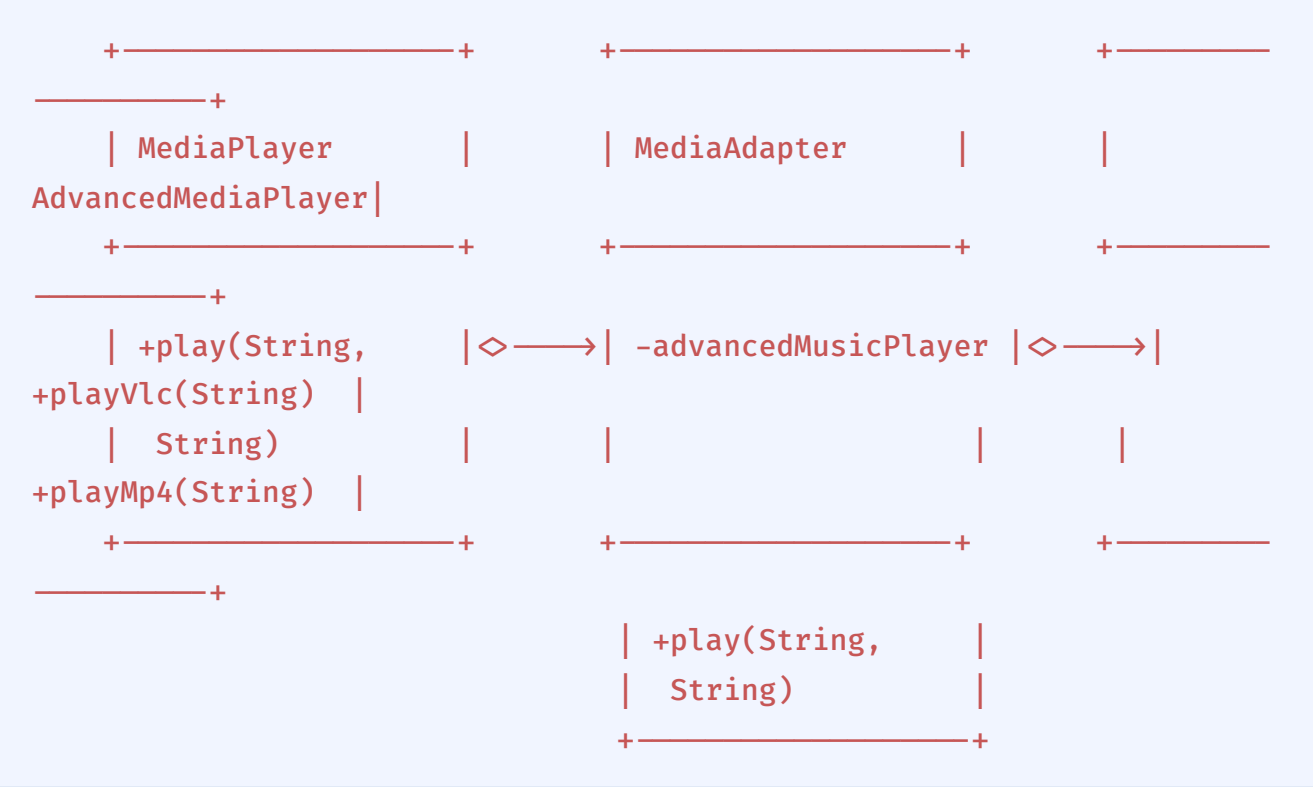
在Java中，适配器模式通常用于将一个类的接口转换成客户期望的另一个接口，使原本接口不兼容的类可以合作无间。下面通过一个具体的例子来介绍适配器模式的应用。

场景描述

假设我们有一个 `MediaPlayer` 接口，它有一个 `play` 方法，用于播放音乐文件。目前它只能播放 `mp3` 格式的文件。现在我们需要扩展功能，使其能够播放 `mp4` 和 `vlc` 格式的文件。但是，我们已经有了一些可以播放这些格式的类（`Mp4Player` 和 `VlcPlayer`），它们的接口与 `MediaPlayer` 不兼容。这时，我们可以使用适配器模式来解决这个问题。

- 目标：`MediaPlayer`
- 被适配者：`AdvancedMediaPlayer`
- 适配器：`MediaAdapter`

类图



代码实现

```java
// MediaPlayer.java
interface MediaPlayer { // 目标
    void play(String audioType, String fileName);
}
// AdvancedMediaPlayer.java
interface AdvancedMediaPlayer { // 被适配者
    void playVlc(String fileName);
    void playMp4(String fileName);
}
// Mp4Player.java
class Mp4Player implements AdvancedMediaPlayer { // 被适配者的具体类
    @Override
    public void playVlc(String fileName) {
        // Do nothing
    }
```

```
    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file. Name: " + fileName);
    }
}
// VlcPlayer.java
class VlcPlayer implements AdvancedMediaPlayer { // 被适配者的具体类
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file. Name: " + fileName);
    }
    @Override
    public void playMp4(String fileName) {
        // Do nothing
    }
}
// MediaAdapter.java
class MediaAdapter implements MediaPlayer { // 适配器
    AdvancedMediaPlayer advancedMusicPlayer;
    public MediaAdapter() {}
    @Override
    public void play(String audioType, String fileName) {
        // Inbuilt support to play mp3 music files
        if (audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing mp3 file. Name: " + fileName);
        }
        // MediaAdapter is providing support to play other file
formats
        else if (audioType.equalsIgnoreCase("vlc")) {
            advancedMusicPlayer = new VlcPlayer();
            advancedMusicPlayer.playVlc(fileName);
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer = new Mp4Player();
            advancedMusicPlayer.playMp4(fileName);
        } else {
            System.out.println("Invalid media. " + audioType + "
format not supported");
        }
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        MediaAdapter audioPlayer = new MediaAdapter();
        audioPlayer.play("mp3", "beyond the horizon.mp3");
        audioPlayer.play("mp4", "alone.mp4");
        audioPlayer.play("vlc", "far far away.vlc");
```

```
            audioPlayer.play("avi", "mind me.avi");
        }
    }
```

输出

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

在这个例子中，`AudioPlayer`类实现了`MediaPlayer`接口，可以播放`mp3`文件。对于`mp4`和`vlc`文件，它使用了一个`MediaAdapter`来适配`AdvancedMediaPlayer`接口，从而实现了播放不同格式文件的功能。这样，我们就通过适配器模式实现了接口的转换，使得原本不兼容的类可以一起工作。

例2

- 用户家里现有一台洗衣机，使用交流电，现在用户新买了一台录音机，录音机只能使用直流电。
- 由于供电系统供给用户家里是交流电，因此用户需要用适配器将交流电转化为直流电供录音机使用
  - 目标（Target）：是名字为DirectCurrent的接口
  - 被适配者（Adaptee）：是名字为AlternateCurrent的接口
  - 适配器：是名字为ElectricAdapter类，该类实现了DirectCurrent接口并包含有AlternateCurrent接口变量。
  - 被适配者（Adaptee）的具体类：PowerCompany

目标（Target）

```
public interface DirectCurrent{
  public String giveDirectCurrent();
}
```

被适配者

```
public interface AlternateCurrent{
  public String giveAlternateCurrent();
}
```

适配器

```
public class ElectricAdapter implements DirectCurrent{
   AlternateCurrent out;
   ElectricAdapter(AlternateCurrent out){
       this.out=out;
   }
   public String giveDirectCurrent(){
     String  m = out.giveAlternateCurrent();  //先由out得到交流电
     StringBuffer str =new StringBuffer(m);
     //以下将交流电转为直流电:
     for(int i=0;i<str.length();i++) {
        if(str.charAt(i)=='0') {
           str.setCharAt(i,'1');
        }
     }
     m =new String(str);
     return m;  //返回直流电
   }
}
```

被适配者的具体类

```
class PowerCompany implements AlternateCurrent { //交流电提供者
    public String giveAlternateCurrent(){
       return "10101010101010101010"; //用这样的串表示交流电
    }
}
```

录音机和洗衣机

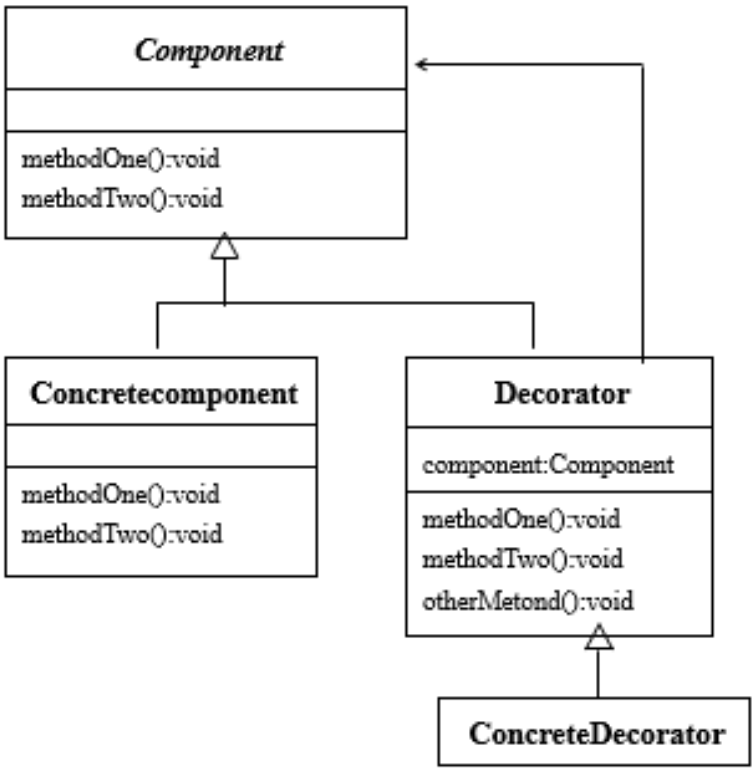
```
class Recorder { //录音机使用直流电
   String name;
   Recorder(){
      name="录音机";
   }
   Recorder(String s){
      name=s;
   }
   public void turnOn(DirectCurrent a){
      String s=a.giveDirectCurrent();
      System.out.println(name+"使用直流电:\n"+s);
      System.out.println("开始录音。");
   }
}
class Wash { //洗衣机使用交流电
   String name;
   Wash(){
     name="洗衣机";
   }
   Wash(String s){
     name=s;
   }
   public void turnOn(AlternateCurrent a){
      String s=a.giveAlternateCurrent();
      System.out.println(name+"使用交流电:\n"+s);
      System.out.println("开始洗衣物。");
   }
}
```

模式的使用

```
public class Application{
   public static void main(String args[]){
      AlternateCurrent aElectric =new PowerCompany(); //交流电
      Wash wash =new Wash();
      wash.turnOn(aElectric); //洗衣机使用交流电
      //对交流电aElectric进行适配得到直流电dElectric:
      DirectCurrent dElectric = new ElectricAdapter(aElectric); //将交流电适配成直流电
      Recorder recorder =new Recorder();
      recorder.turnOn(dElectric); //录音机使用直流电
   }
}
```

```
洗衣机使用交流电:
10101010101010101010
开始洗衣物。
录音机使用直流电:
11111111111111111111
开始录音。
```

装饰模式



- 抽象组件（Component）：抽象组件（是抽象类）定义了需要进行装饰的方法。抽象组件就**是“被装饰者”角色。**
- 具体组件（ConcreteComponent）：具体组件是抽象组件的一个子类。
- 装饰（Decorator）：该角色是抽象组件的一个子类，**是“装饰者”角色**，其作用是装饰具体组件。Decorator角色需要**包含抽象组件的引用。**
- 具体装饰（ConcreteDecotator）：具体装饰是Decorator角色的一个非抽象子类

给麻雀安装智能电子翅膀，未来我们可能对麻雀做进一步装饰。

- 抽象组件的名字是Bird；
- 具体组件角色：Sparrow类，该类的实例模拟麻雀；
- 装饰（Decorator）角色是抽象组件Bird的一个子类，需要包含被装饰者（抽象组件）的引用；
- 具体装饰是SparrowDecorator类，该类使用eleFly()方法去装饰fly()方法。

抽象组件

Bird.java

```
public abstract class Bird{
   public abstract int fly();
}
```

具体组件

```
public class Sparrow extends Bird{
   public final int DISTANCE=100;
   public int fly(){
      return DISTANCE;
   }
}
```

装饰

```
public abstract class Decorator extends Bird{
   Bird bird;      //被装饰者
   public Decorator(){
   }
   public Decorator(Bird bird){
      this.bird=bird;
   }
   public abstract int eleFly();//用于装饰fly()的方法,行为由具体装饰者去实现
}
```

具体装饰

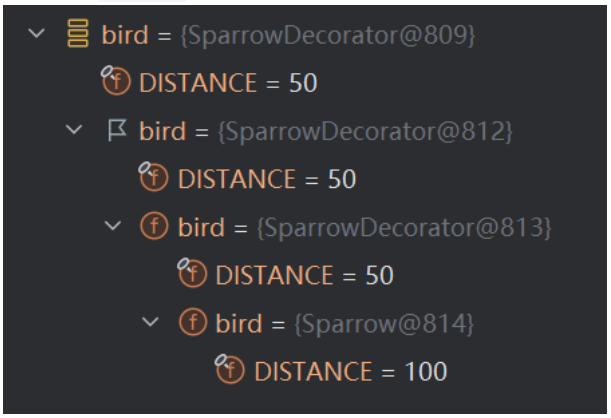
```
public class SparrowDecorator extends Decorator{
   public final int DISTANCE=50;  //eleFly方法(模拟电子翅膀)能飞50米
   SparrowDecorator(Bird bird){
      super(bird);
   }
   public int fly(){   //被装饰的方法
      int distance=0;
      distance=bird.fly()+eleFly();//让装饰者bird首先调用fly()，然后再调用eleFly()
      return distance;
   }
   public int eleFly(){      //具体装饰者重写装饰者中用于装饰的方法
      return DISTANCE;
   }
}
```

模式的使用

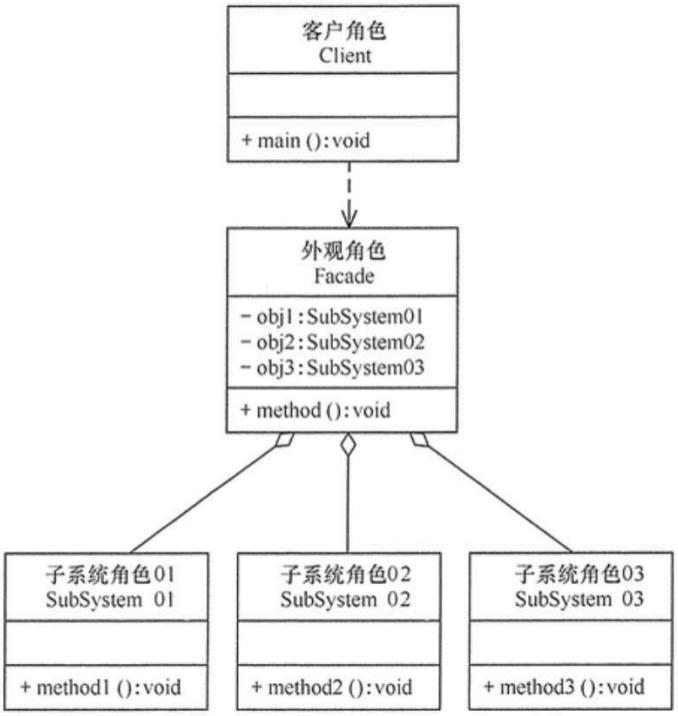
```
public class Application{
   public static void main(String args[]){
      Bird bird=new Sparrow();
      System.out.println("没有安装电子翅膀的小鸟飞行距离:"+bird.fly());
      bird=new SparrowDecorator(bird); //bird通过"装饰"安装了1个电子翅膀
      System.out.println("安装1个电子翅膀的小鸟飞行距离:"+bird.fly());
      bird=new SparrowDecorator(bird); //bird通过"装饰"安装了2个电子翅膀
      System.out.println("安装2个电子翅膀的小鸟飞行距离:"+bird.fly());
      bird=new SparrowDecorator(bird); //bird通过"装饰"安装了3个电子翅膀
      System.out.println("安装3个电子翅膀的小鸟飞行距离:"+bird.fly());
   }
}
```

```
没有安装电子翅膀的小鸟飞行距离:100
安装1个电子翅膀的小鸟飞行距离:150
安装2个电子翅膀的小鸟飞行距离:200
安装3个电子翅膀的小鸟飞行距离:250
```

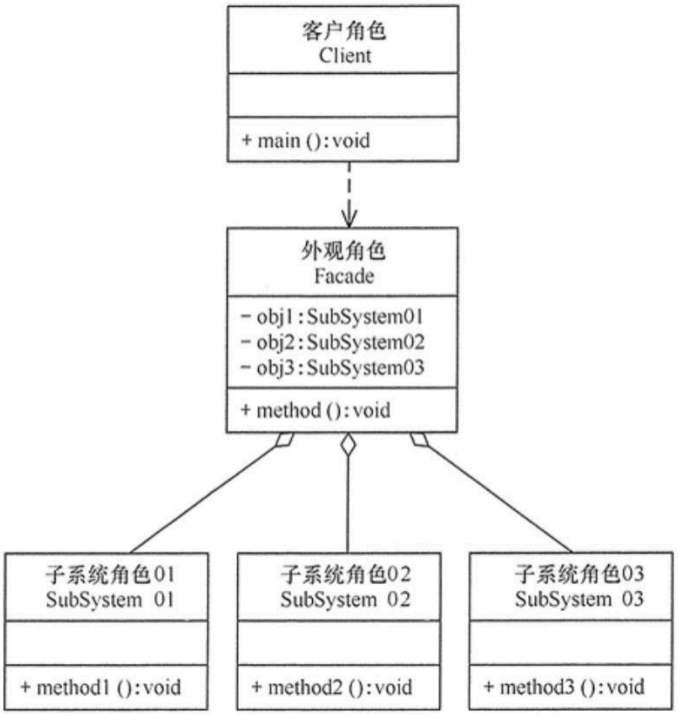
最后的`bird`



外观模式（门面模式）



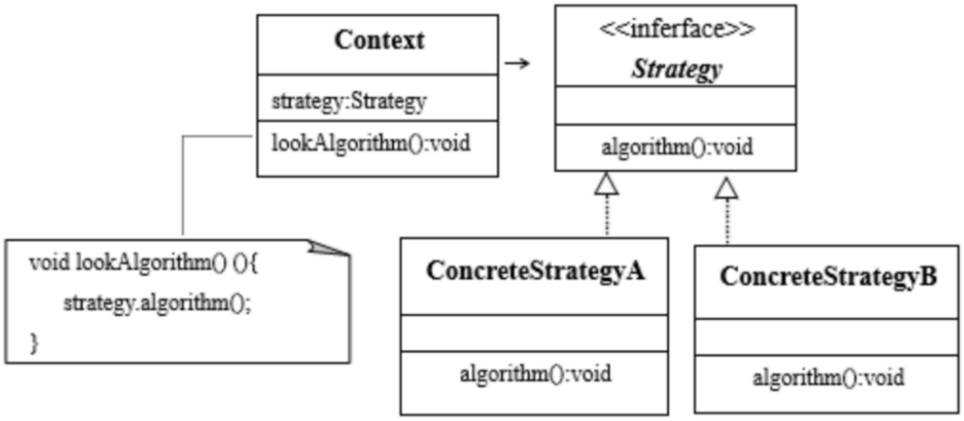
- 外观（Facade）模式又叫作门面模式，是一种通过为多个复杂的子系统提供一个**一致的接口**，而使这些子系统**更加容易被访问**的模式。
- 该模式对外有一个统一接口，外部应用程序不用关心内部子系统的具体细节，这样会大大降低应用程序的复杂度，降低其与子系统的耦合，提高了程序的可维护性。
- 是“**迪米特法则**”的典型应用
  迪米特法则: 一个软件实体应当尽可能少地与其他实体发生相互作用


客户角色
Client
+ main ():void
外观角色
Facade
- obj1 : SubSystem01
- obj2 : SubSystem02
- obj3 : SubSystem03
+ method ():void
子系统角色01
SubSystem 01
+ method1 ():void
子系统角色02
SubSystem 02
+ method2 ():void
子系统角色03
SubSystem 03
+ method3 ():void

```
//子系统角色
class SubSystem01 {
    public void method1() {
        System.out.println("子系统01的method1()被调用！");
    }
}

//子系统角色
class SubSystem02 {
    public void method2() {
        System.out.println("子系统02的method2()被调用！");
    }
}

//子系统角色
class SubSystem03 {
    public void method3() {
        System.out.println("子系统03的method3()被调用！");
    }
}
```

```
public class FacadePattern {
    public static void main(String[] args) {
        Facade f = new Facade();
        f.method();
    }
}

//外观角色
class Facade {
    private SubSystem01 obj1 = new SubSystem01();
    private SubSystem02 obj2 = new SubSystem02();
    private SubSystem03 obj3 = new SubSystem03();

    public void method() {
        obj1.method1();
        obj2.method2();
        obj3.method3();
    }
}
```

策略模式



- **策略**（Strategy）：策略是一个**接口**，该接口定义若干个算法标识，即**定义了若干个抽象方法。**核心就是将类中经常需要变化的部分分割出来，并将每种可能的变化对应地交给抽象类的一个子类或实现接口的一个类去负责，从而让类的设计者不去关心具体实现，避免所设计的类依赖于具体的实现。
- **上下文**（Context）：上下文是依赖于策略接口的**类**（是面向策略设计的类），即上下文**包含有用策略声明的变量。**上下文中提供一个**方法**，该方**法委托策略变量调用具体策略所实现的策略接口中的方法。**
- **具体策略**（ConcreteStrategy）：具体策略是**实现策略接口的类。**具体策略实现策略接口所定义的抽象方法，即**给出算法标识的具体算法。**

问题：在多个裁判负责打分的比赛中，每位裁判给选手一个得分，选手的最后得分是根据全体裁判的得分计算出来的。请给出几种计算选手得分的评分方案（策略），对于某次比赛，可以从你的方案中选择一种方案作为本次比赛的评分方案。

- 在这里我们把策略接口命名为：`Strategy`。在具体应用中，这个角色的名字可以根据具体问题来命名。
- 在本问题中将上下文命名为 `AverageScore`，即让 `AverageScore` 类依赖于 `Strategy` 接口。
- 每个具体策略负责一系列算法中的一个。
- 策略（ `Strategy` ）

```
public interface Strategy {
   public double computerAverage(double [] a);
}
```

- 上下文（`Context`）

```
public class  AverageScore{
    Strategy strategy;
    public void setStrategy(Strategy strategy){
        this.strategy=strategy;
    }
    public double getAverage (double [] a){
      if(strategy!=null)
        return strategy.computerAverage(a);
      else {
        System.out.println("没有求平均值的算法,得到的-1不代表平均值");
        return -1;
      }
    }
}
```

- 具体策略StrategyA.java

```
public class StrategyA implements Strategy{
   public double computerAverage(double [] a){
      double score=0,sum=0;
      for(int i=0;i<a.length;i++){
         sum=sum+a[i];
      }
      score=sum/a.length;
      return score;
   }
}
```

- 具体策略StrategyB.java

```
import java.util.Arrays;
public class StrategyB implements Strategy{
    public double computerAverage(double [] a){
        if(a.length<=2)
            return 0;
        double score=0,sum=0;
        Arrays.sort(a);   //排序数组
        for(int i=1;i<a.length-1;i++){
          sum=sum+a[i];
        }
        score=sum/(a.length-2);
        return score;
    }
}
```

- 模式的使用

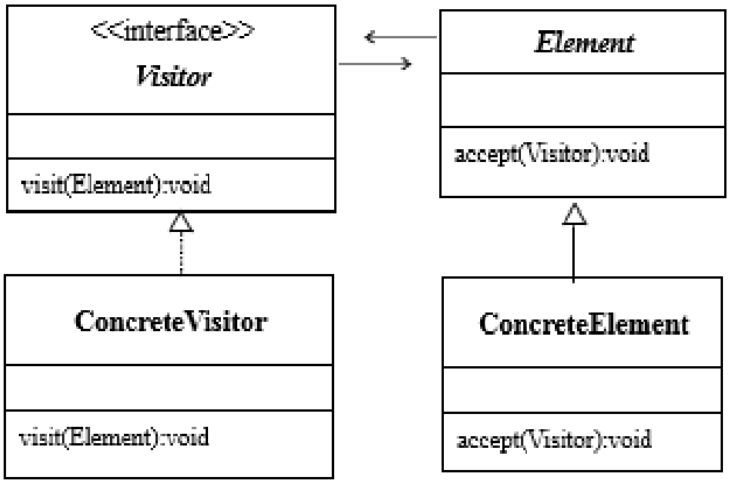
```
class Person{
   String name;
   double score;
   public void setScore(double t){
      score=t;
   }
   public void setName(String s){
      name=s;
   }
   public double getScore(){
      return score;
   }
   public String getName(){
      return name;
   }
}
```

```
public class Application{
   public static void main(String args[]){
     AverageScore game=new AverageScore();//上下文对象game
     game.setStrategy(new StrategyA());  //上下文对象使用具体策略
     Person zhang=new Person();
     zhang.setName("张三");
     double [] a={9.12,9.25,8.87,9.99,6.99,7.88};
     double aver = game.getAverage(a) ;   //上下文对象得到平均值
     zhang.setScore(aver);
     System.out.println("算法A:");
     System.out.printf("%s最后得分:%5.3f%n",zhang.getName(),zhang.getScore());
     game.setStrategy(new StrategyB());
     aver = game.getAverage(a) ;          //上下文对象得到平均值
     zhang.setScore(aver);
     System.out.println("算法B:");
     System.out.printf("%s最后得分:%5.3f%n",zhang.getName(),zhang.getScore());
  }
}
```

```
算法A:
张三最后得分:8.683
算法B:
张三最后得分:8.780
```

访问者模式

- **模式优点**：在不改变一个集合中的元素的类的情况下，可以增加新的施加于该元素上的新操作。保持一定的扩展性。
- **使用场景**：需要对集合中的对象进行很多不同的并且不相关的操作，而我们又不想修改对象的类，就可以使用访问者模式。访问者模式可以在Visitor类中集中定义一些关于集合中对象的操作。



- 抽象元素（Element）：一个抽象类，该类定义了接收访问者的accept操作。
- 具体元素（Concrete Element）：Element的子类。
- 抽象访问者（Visitor）：一个接口，该接口定义操作具体元素的方法。
- 具体访问者（Concrete Visitor）：实现Visitor接口的类。
- **门诊部**是一个类似于**访问者的对象**，它可以**访问不同类型的病人对象**，例如普通病人、急诊病人、儿科病人等。
- **不同类型的病人对象可以有不同的处理方法**，例如看病、输液、检查等。
- 门诊部可以对不同类型的病人对象进行不同的操作，而不需要改变病人对象的类层次结构。
- 抽象访问者

```
interface Visitor {
    void visit(Patient patient);
}
```

- 具体访问者

```
class Doctor implements Visitor {
    @Override
    public void visit(Patient patient) {
        System.out.println("医生为病人：" +
patient.getName() + " 处理 " + patient.getIllness());
    }
}
```

- 抽象元素

```
interface Element {
    void accept(Visitor visitor);
}
```

- 具体元素

```
class Patient implements Element {
    private String name;
    private String illness;

    public Patient(String name, String illness) {
        this.name = name;
        this.illness = illness;
    }

    public String getName() {
        return name;
    }

    public String getIllness() {
        return illness;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

- 结构对象

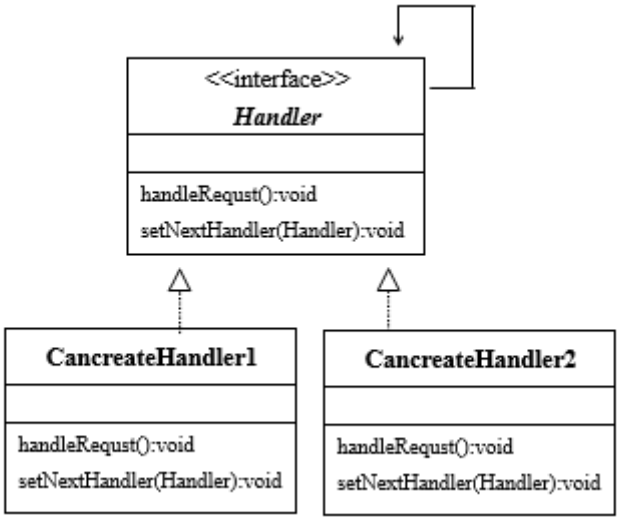
```
class ObjectStruture {
    private List<Element> elements = new ArrayList<>();

    public void add(Element element) {
        elements.add(element);
    }

    public void remove(Element element) {
        elements.remove(element);
    }

    public void accept(Visitor visitor) {
        for (Element element : elements) {
            element.accept(visitor);
        }
    }
}
```

- 测试案例

```
public class Test {
    public static void main(String[] args) {
        Patient p1 = new Patient("张三", "感冒");
        Patient p2 = new Patient("李四", "发烧");

        ObjectStruture objectStruture = new ObjectStruture();
        objectStruture.add(p1);
        objectStruture.add(p2);

        Doctor doctor = new Doctor();
        objectStruture.accept(doctor);
    }
```

责任链模式

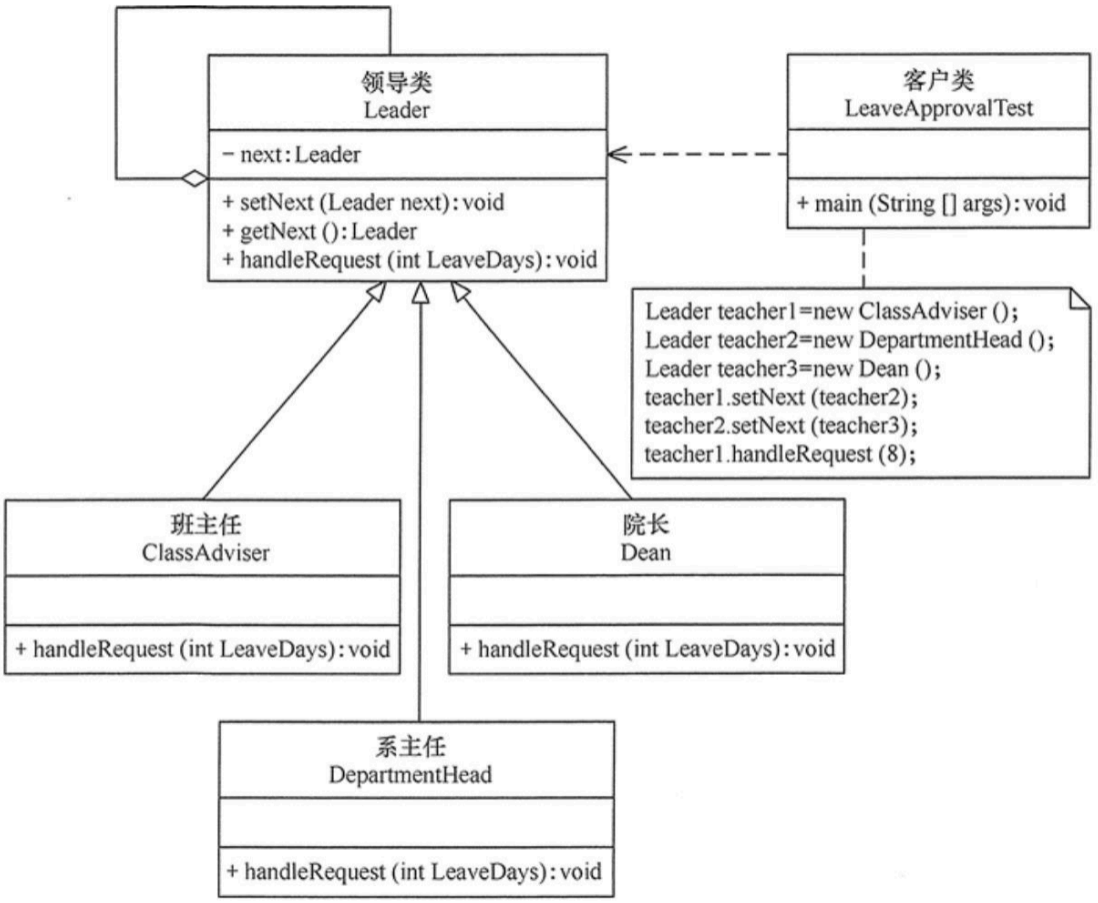
责任链模式是使用多个对象处理用户请求的成熟模式，责任链模式的关键是将用户的请求分派给许多对象。



- 处理者（Handler）：处理者是一个接口，负责规定具体处理者**处理用户的请求的方法**以及具体处理者**设置后继对象**的方法。
- 具体处理者（ConcreteHandler）：具体处理者是实现处理者接口的类的实例。具体处理者通过调用处理者接口规定的方法处理用户的请求，即在接到用户的请求后，处理者将调用接口规定的方法，在执行该方法的过程中，**如果发现能处理用户的请求，就处理有关数据，否则就反馈无法处理的信息给用户，然后将用户的请求传递给自己的后继对象。**

案例：用责任链模式设计一个请假条审批模块

- 假如规定学生请假小于或等于 2 天，班主任可以批准；小于或等于 7 天，系主任可以批准；小于或等于 10 天，院长可以批准；其他情况不予批准；



抽象处理者：领导类

```
//抽象处理者：领导类
abstract class Leader {
    private Leader next;

    public void setNext(Leader next) {
        this.next = next;
    }

    public Leader getNext() {
        return next;
    }

    //处理请求的方法
    public abstract void handleRequest(int LeaveDays);
}
```

具体处理者1：班主任类

```
//具体处理者1：班主任类
class ClassAdviser extends Leader {
    public void handleRequest(int LeaveDays) {
        if (LeaveDays <= 2) {
            System.out.println("班主任批准您请假" + LeaveDays + "天。");
        } else {
            if (getNext() != null) {
                getNext().handleRequest(LeaveDays);
            } else {
                System.out.println("请假天数太多，没有人批准该假条！");
            }
        }
    }
}
```

具体处理者2：系主任类

```
//具体处理者2：系主任类
class DepartmentHead extends Leader {
    public void handleRequest(int LeaveDays) {
        if (LeaveDays <= 7) {
            System.out.println("系主任批准您请假" + LeaveDays + "天。");
        } else {
            if (getNext() != null) {
                getNext().handleRequest(LeaveDays);
            } else {
                System.out.println("请假天数太多，没有人批准该假条！");
            }
        }
    }
}
```

具体处理者：院长类

```
//具体处理者3：院长类
class Dean extends Leader {
    public void handleRequest(int LeaveDays) {
        if (LeaveDays <= 10) {
            System.out.println("院长批准您请假" + LeaveDays + "天。");
        } else {
            if (getNext() != null) {
                getNext().handleRequest(LeaveDays);
            } else {
                System.out.println("请假天数太多，没有人批准该假条！");
            }
        }
    }
}
```

测试类

```
public class LeaveApprovalTest {
    public static void main(String[] args) {
        //组装责任链
        Leader teacher1 = new ClassAdviser();
        Leader teacher2 = new DepartmentHead();
        Leader teacher3 = new Dean();
        //Leader teacher4=new DeanOfStudies();
        teacher1.setNext(teacher2);
        teacher2.setNext(teacher3);
        //teacher3.setNext(teacher4);
        //提交请求
        teacher1.handleRequest(8);
    }
}
```

dlc：具体处理者4：教务处长类

```
//具体处理者4：教务处长类
class DeanOfStudies extends Leader {
    public void handleRequest(int LeaveDays) {
        if (LeaveDays <= 20) {
            System.out.println("教务处长批准您请假" + LeaveDays + "天。");
        } else {
            if (getNext() != null) {
                getNext().handleRequest(LeaveDays);
            } else {
                System.out.println("请假天数太多，没有人批准该假条！");
            }
        }
    }
}
```

观察者模式

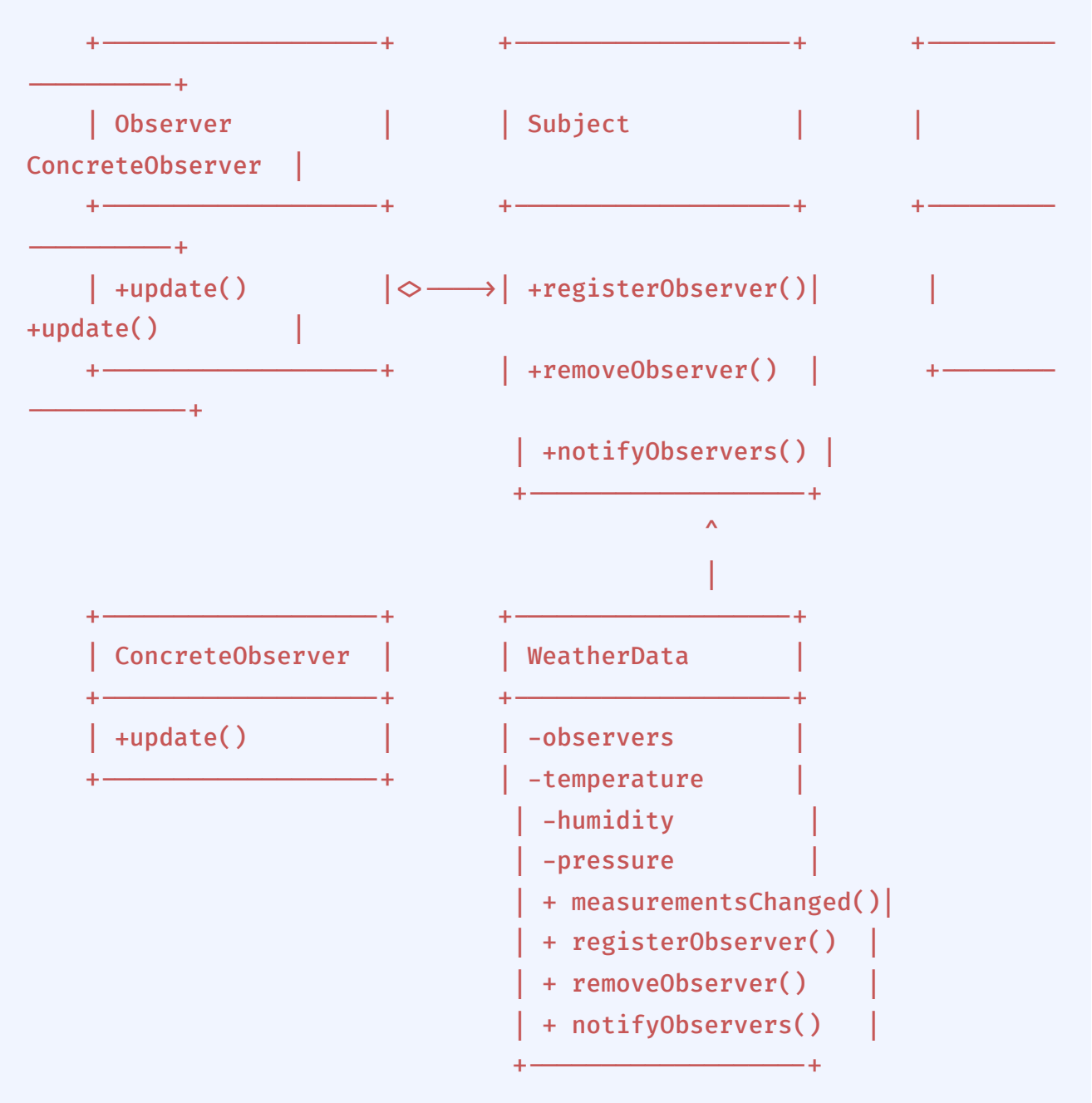
被观察者存了一个list表示观察者，观察者存了自己观察的对象
当被观察者发生变化时，通知观察者，观察者更新数据并展示出来

观察者模式（Observer Pattern）是一种行为设计模式，它定义了对象之间的一对多依赖关系，当一个对象改变状态时，所有依赖于它的对象都会得到通知并自动更新。这种模式在Java中经常用于实现事件处理系统、消息订阅和发布等场景。

场景描述

假设我们有一个天气数据类（`WeatherData`），它包含了温度、湿度等天气信息。我们希望当天气数据更新时，能够通知多个显示天气信息的界面（如当前天气状况显示、天气统计信息显示等）进行更新。这里可以使用观察者模式来实现。

类图



代码实现

```
// Observer.java 观察者
public interface Observer {
    void update(float temperature, float humidity, float pressure);
}
// Subject.java 被观察者
public interface Subject {
    void registerObserver(Observer o);
```

```
    void removeObserver(Observer o);
    void notifyObservers();
}
// DisplayElement.java
public interface DisplayElement {
    void display();
}
// WeatherData.java 被观察者——天气
public class WeatherData implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;
    public WeatherData() {
        observers = new ArrayList<>();
    }
    public void registerObserver(Observer o) {
        observers.add(o);
    }
    public void removeObserver(Observer o) {
        observers.remove(o);
    }
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }
    public void measurementsChanged() {
        notifyObservers();
    }
    public void setMeasurements(float temperature, float humidity,
float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
    // Other WeatherData methods here.
}
// CurrentConditionsDisplay.java 观察者，实现展示功能
public class CurrentConditionsDisplay implements Observer,
DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;
    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
```

```
    }
    public void update(float temperature, float humidity, float
pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }
    public void display() {
        System.out.println("Current conditions: " + temperature + "F
degrees and " + humidity + "% humidity");
    }
}
// Main.java
public class Main {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();
        CurrentConditionsDisplay currentDisplay = new
CurrentConditionsDisplay(weatherData);
        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

输出

```
Current conditions: 80.0F degrees and 65.0% humidity
Current conditions: 82.0F degrees and 70.0% humidity
Current conditions: 78.0F degrees and 90.0% humidity
```

在这个例子中，`WeatherData`类实现了`Subject`接口，它有一个观察者列表，用于注册、移除和通知观察者。`CurrentConditionsDisplay`类实现了`Observer`和`DisplayElement`接口，它注册为`WeatherData`的观察者，并在数据更新时接收通知并显示当前天气状况。
当`WeatherData`的`setMeasurements`方法被调用时，它会更新天气数据并调用`measurementsChanged`方法，后者会通知所有注册的观察者。观察者接收到通知后，会调用它们的`update`方法来获取新的数据并更新显示。
这样，我们就通过观察者模式实现了当天气数据变化时，自动通知

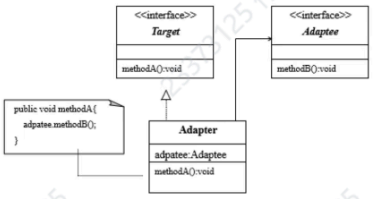
other

- 原型模式

用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象

实现了一个原型接口，该接口用于创建当前对象的克隆

结构型



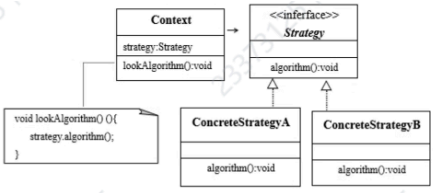
- 适配器模式

将一个类的接口转换成客户希望的另外一个接口实现了一个原型接口，该接口用于创建当前对象的克隆

适配器模式中包括三种角色：

- **目标（Target）**：目标是一个接口，该接口是客户想使用的接口。
- **被适配者（Adaptee）**：被适配者是一个已经存在的接口或抽象类，这个接口或抽象类需要适配。
- **适配器（Adapter）**：适配器是一个类，该类实现了目标接口并包含有被适配者的引用，即适配器的职责是对被适配者接口（抽象类）与目标接口进行适配。

行为型



- 策略模式

核心就是将类中经常需要变化的部分分割出来，并将每种可能的变化对应地交给抽象类的一个子类或实现接口的一个类去负责，从而让类的设计者不去关心具体实现

结构中包含以下三种角色：

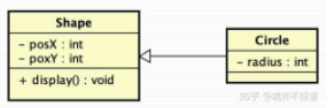
- **策略（Strategy）**：策略是一个接口，该接口定义若干个算法标识，即定义了若干个抽象方法。
- **具体策略（ConcreteStrategy）**：具体策略是实现策略接口的类。具体策略实现策略接口所定义的抽象方法，即给出算法标识的具体算法。
- **上下文（Context）**：上下文是依赖于策略接口的类（是面向策略设计的类），即上下文包含有用策略声明的变量。上下文中提供一个方法，该方法委托策略变量调用具体策略所实现的策略接口中的方法。

UML图 7

• **泛化**

是一种**继承关系**，表示子类继承父类的所有特征和行为

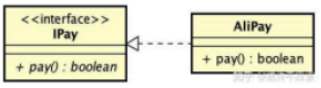
带**空心三角箭头的实线**，箭头指向父类



• **实现**

是一种类与接口的关系，表示类是接口所有特征和行为的实现

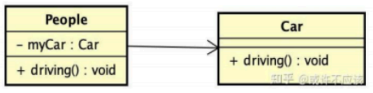
带**空心三角箭头的虚线**，箭头指向接口



• **关联**

是一种拥有关系，它使得一个类知道另一个类的属性和方法

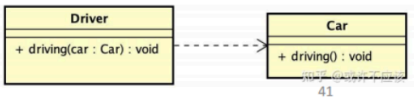
带**普通箭头的实线**，指向被拥有者。双向的关联可以有两个箭头，或者没有箭头。单向的关联有一个箭头



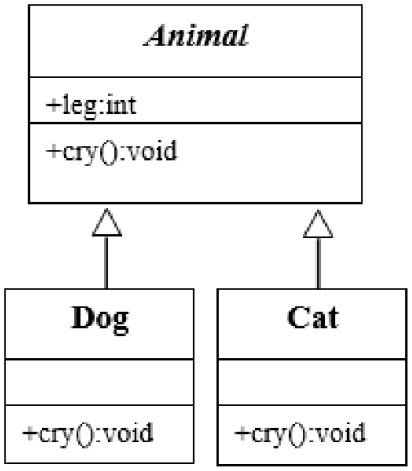
•**依赖**

是一种使用关系，即一个类的实现需要另一个类的协助

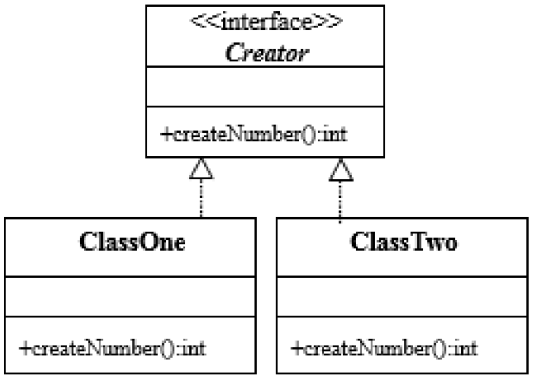
带**普通箭头的虚线**，普通箭头指向被使用者



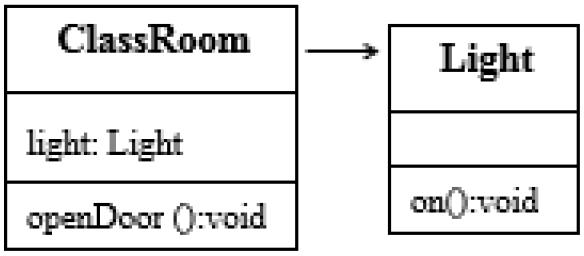
- 泛化：继承



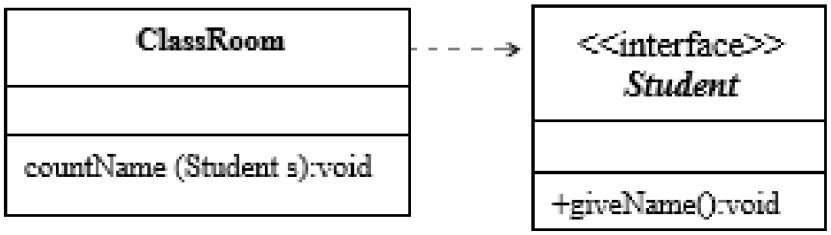
- 实现



- 关联/组合（比依赖强）：“拥有”



- 依赖：方法中”用到“



Java集合框架 8

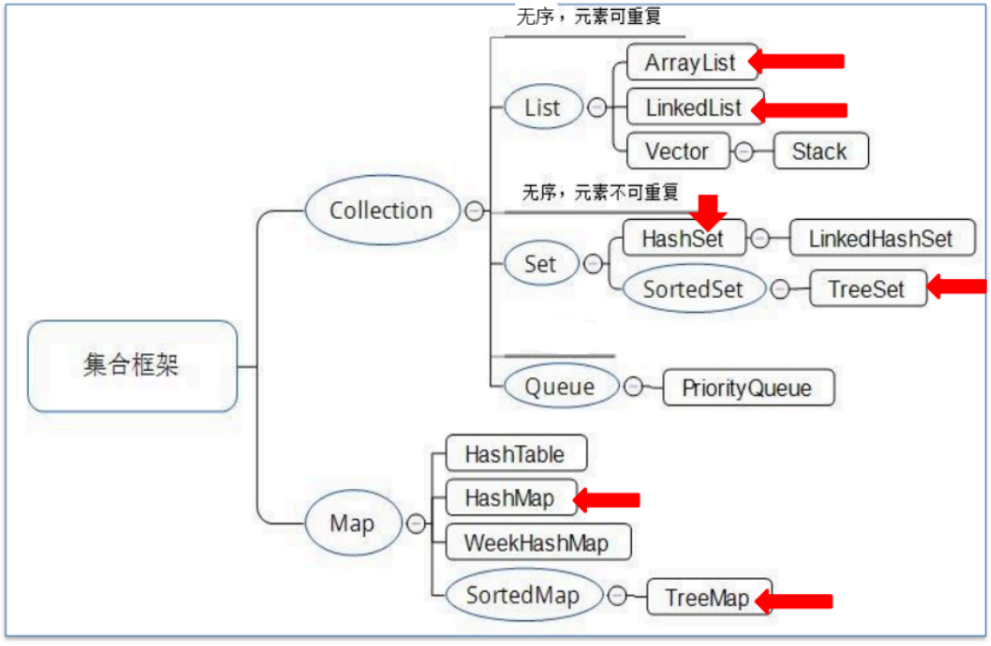
所有集合类都位于 java.util 包下。Java的集合类主要由两个接口派生而出：Collection 和 Map

Set、 List 和 Map 可以看做集合的三大类：

Set 接口继承 Collection，无序集合，集合中的元素不可以重复

List 接口继承 Collection，允许重复，插入序

Map 集合中保存 key-value 对形式的元素



普通数组的定义：`int[] a = new int[10];`

ArrayList：无序，可重复，长度可变，**遍历元素和随机访问元素效率较高**

ArrayList

ArrayList<String> sites = new ArrayList<String>();

添加元素：sites.add(“str”)

访问元素：sites.get(index)

修改元素：sites.set(2, "Wiki");

删除元素：sites.remove(index);

使用Collections排序：Collections.sort(sites); // 默认数字/字母排序

数组大小：`site.size()`

LinkedList：无序，可重复，FIFO，**插入删除元素效率较高**

方法名	说　明
void addFirst(Object o)	在列表的首部添加元素
void addLast(Object o)	在列表的末尾添加元素
Object getFirst()	返回列表中的第一个元素
Object getLast()	返回列表中的最后一个元素
Object removeFirst()	删除并返回列表中的第一个元素
Object removeLast()	删除并返回列表中的最后一个元素

```
public class Test3 {
    public static void main(String[] args) {
        Dog ououDog = new Dog("欧欧", "雪娜瑞");
        Dog yayaDog = new Dog("亚亚", "拉布拉多");
        Dog meimeiDog = new Dog("美美", "雪娜瑞");
        Dog feifeiDog = new Dog("菲菲", "拉布拉多");

        LinkedList<Object> dogs = new LinkedList<Object>();
//      LinkedList<Dog> dogs = new LinkedList<Dog>();
        dogs.add(ououDog);
        dogs.add(yayaDog);
        dogs.addFirst(meimeiDog);  //// 添加meimeiDog到指定位置
        dogs.addLast(feifeiDog); // 添加feifeiDog到指定位置

        System.out.println("共计有" + dogs.size() + "条狗狗。");

        System.out.println("分别是：");
        for (int i = 0; i < dogs.size(); i++) {
            Dog dog = (Dog) dogs.get(i);
            System.out.println(dog.getName() + "\t" + dog.getStrain());
        }
```

```
Dog dogFirst= (Dog)dogs.getFirst();
System.out.println("第一条狗狗昵称是"+dogFirst.getName() );

Dog dogLast= (Dog)dogs.getLast();
System.out.println("最后一条狗狗昵称是"+dogLast.getName());

dogs.removeFirst();
dogs.removeLast();
System.out.println("共计有" + dogs.size() + "条狗狗。");

System.out.println("分别是：");
for (int i = 0; i < dogs.size(); i++) {
    Dog dog = (Dog) dogs.get(i);
    System.out.println(dog.getName() + "\t" + dog.getStrain());
}
```

HashSet：无序，不可重复

HashSet

HashSet<String> sites = new HashSet<String>();

添加元素：sites.add(“str”)

判断元素是否存在：sites.contains("success")

删除元素：sites.remove("success");

使用Collections排序：先转成List再用ArrayList的方法排序

List<String> sortedList = new ArrayList<>(sites);

*也可以是key-value的，比如HashSet<String, String>，第一个做key，第二个做value

HashMap：无序，键（Key）不能重复，值（Value）可以重复

方法名	说　明
Object put(Object key, Object val)	以“键-值对”的方式进行存储
Object get (Object key)	根据键返回相关联的值，如果不存在指定的键，返回null
Object remove (Object key)	删除由指定的键映射的“键-值对”
int size()	返回元素个数
Set keySet ()	返回键的集合
Collection values ()	返回值的集合
Boolean containsKey (Object key)	如果存在由指定的键映射的“键-值对”，返回true

重写排序

如果 `a` 是 list：`Collection.sort(a)`

如果 `a` 是普通数组：`Arrays.sort(a)`

```
class Person implements Comparable<Person>{
    String name;
    int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
@Override
    public int compareTo(Person person) {
        return this.age - person.age;
    }
}
```

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
       Person[] persons
= new Person[]{new Person("tom", 20), new Person("jack", 12)};
       System.out.print("排序之前： ");
       for (Person p : persons) {
          System.out.print("name:" + p.name + ", age:" + p.age + " ");
       }
       // 排序之后
       Arrays.sort(persons);
       System.out.println("排序之后:");
       for (Person p : persons) {
          System.out.print("name:" + p.name + ", age:" + p.age + " ");
       }
```


例

1. 关于Java类LinkedList的特点，下面描述正确的是（）_____
A 查询快
B 增删快
C 元素不重复
D 元素自然排序

正确答案: B

10. 下面Java中关于List、Set的说法正确的是（）_____
A List 接口存储一组唯一，有序的对象
B Set 接口存储一组唯一，有序的对象
C Set可以允许插入多个null值
D ArrayList和Vector都继承自List

正确答案: D