# Java基础(Java SE)

接口、内部类



# 本章目标

- 掌握多态的优势和应用场合
- 掌握父类和子类之间的类型转换
- 掌握instanceof运算符的使用
- 使用父类作为方法形参实现多态
- 掌握接口基础知识
- 掌握接口作为一种约定和能力的含义
- 掌握匿名内部类

# 什么是多态

- 生活中的多态
  - 你能列举出一个多态的生活示例吗?

    同一种事物，由于条件不同，产生的结果也不同

- 程序中的多态

  多态：同一个引用类型，使用不同的实例而执行不同操作

  父类引用，子类对象

# 多态性

- 多态-----子类的对象可以作为父类对象使用
- 一个引用类型变量可能指向其子类对象
- 一个对象只能有一种确定的数据类型
- 一个引用类型变量如果声明为父类的类型，但实际引用的是子类对象，那么该变量就不能再访问子类中添加的属性和方法

# 动态绑定

- java运行时系统根据调用该方法的实例，来决定调用哪个方法。对子类的一个实例，如果子类重写了父类的方法，则运行时系统调用子类的方法；如果子类继承了父类的方法（未重写），则运行时系统调用父类的方法。

- 编译时类型和运行时类型

- Java中的引用变量有两个类型：一个是编译时的类型，一个是运行时的类型，编译时的类型由声明该变量时使用的类型决定，运行时的类型由实际赋给该变量的对象决定。如果编译时的类型与运行时的类型不一致就会出现多态。

# 如何实现多态3–1

- 用多态实现打印机
  - 分为黑白打印机和彩色打印机
  - 不同类型的打印机打印效果不同

# 如何实现多态3-2

- 使用多态实现思路
  - 编写父类
  - 编写子类，子类重写父类方法
  - 运行时，使用父类的类型，子类的对象

# 如何实现多态3-3

## 编码实现

父类

```
abstract class Printer{
    print(String str);
}
```

实现多态的两个要素：
1.方法重写
2.使用父类类型

同一种操作方式，不同的操作对象

子类

```
class ColorPrinter {
    print(String str) {
        System.out.println("输出彩色的"+str);
    }
}
```

```
class BlackPrinter {
    print(String str) {
        System.out.println("输出黑白的"+str);
    }
}
```

运行

```
public static void main(String[] args) {
    Printer p = new ColorPrinter();
    p.print();
    p = new BlackPrinter();
    p.print();
}
```

只能调用父类已经定义的方法

# 方法重写

- 方法重写的规则
  - 在继承关系的子类中
  - 重写的方法名、参数、返回值类型必须与父类相同
  - 私有方法不能继承因而也无法重写

| | 位置 | 方法名 | 参数表 | 返回值 | 访问修饰符 |
|---|---|---|---|---|---|
| 方法重写 | 子类 | 相同 | 相同 | 相同 | 不能比父类更严格 |
| 方法重载 | 同类 | 相同 | 不同 | 无关 | 无关 |

# 对象造型（Casting ）

- 对Java对象的强制类型转换称为造型
  - 从子类到父类的类型转换称为up-casting。这种转换是安全的，可以自动进行
  - 从父类到子类的类型转换称为down-casting。必须通过造型(强制类型转换)实现。这种转换是不安全的，在转换前可以使用instanceof操作符测试一个对象的类型
  - 无继承关系的引用类型间的转换是非法的

# instanceof运算符

对象 instanceof 类或接口

- 该运算符用来判断一个对象是否属于一个类或者实现了一个接口，结果为true或false

- 在强制类型转换之前通过instanceof运算符检查对象的真实类型，可以避免类型转换异常，从而提高代码健壮性

```
public class TestPoly2 {
    public static void main(String[] args) {
        Pet pet = new Penguin("楠楠", "Q妹");
        // Pet pet = new Dog("欧欧", "雪娜瑞");
        pet.eat();
        if (pet instanceof Dog) {
            Dog dog = (Dog) pet;
            dog.catchingFlyDisc();
        } else if (pet instanceof Penguin) {
            Penguin pgn = (Penguin) pet;
            pgn.swimming();
        }
    }
}
```

# 为什么使用接口

- 只有抽象方法的抽象类?

  可以用接口来表示

- 用接口代替这样的抽象类，是因为：

  接口有比抽象类更好的特性：
  1.可以被多继承
  2.设计和实现完全分离
  3.更自然的使用多态
  4.更容易搭建程序框架
  5.更容易更换实现

# 什么是接口

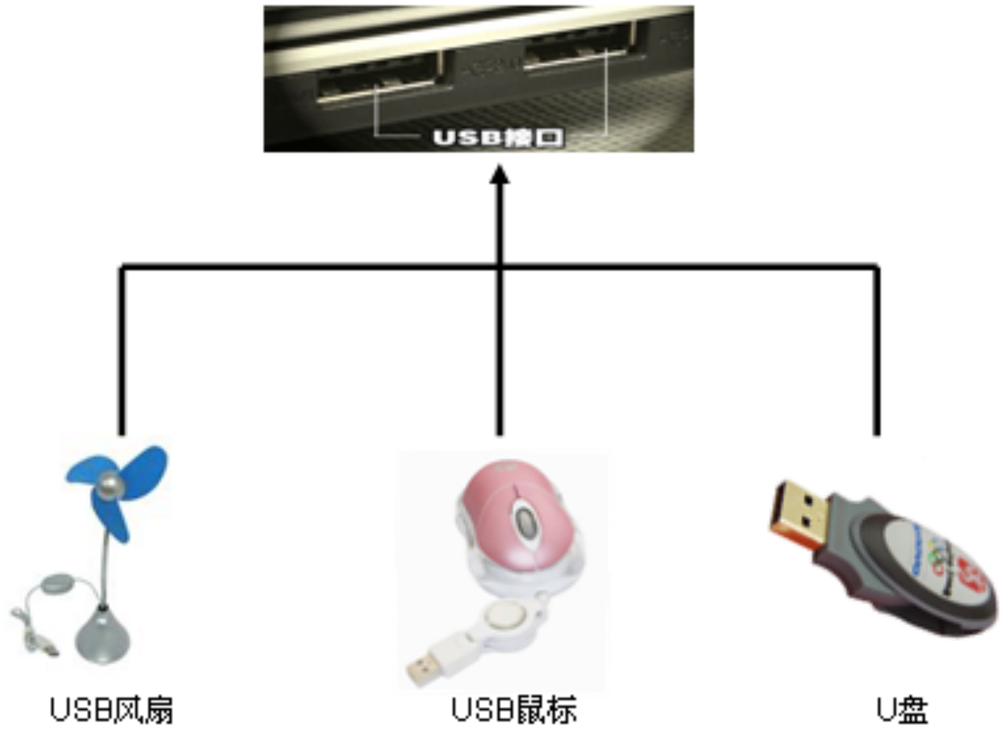
- 认识一下接口

```
public interface MyInterface {
  void print();

}
```

- 必须知道的接口特性
  - 接口不可以被实例化　　常作为类型使用
  - 实现类必须实现接口的所有方法　　抽象类除外
  - 实现类可以实现多个接口　　Java中的多继承
  - 接口中的变量都是静态常量　　所有方法都是:public abstract

# 如何使用接口

▷ 用程序描述USB接口



# 如何使用接口

- 可以使用Java接口来实现
  - USB接口本身没有实现任何功能
  - USB接口规定了数据传输的要求
  - USB接口可以被多种USB设备实现

| | |
|---|---|
| 编写USB接口 | 根据需求设计方法 |
| 实现USB接口 | 实现所有方法 |
| 使用USB接口 | 用多态的方式使用 |

# 接口是一种约定

- 生活中，我们使用的两相电源插座，规定了：
  - 两个接头间的额定电压
  - 两个接头间的距离
  - 接头的形状
- 接口是一种约定

有些接口只有名称

  - 体现在接口名称和注释上
- 面向接口编程

方法的实现方式要通过注释来约定

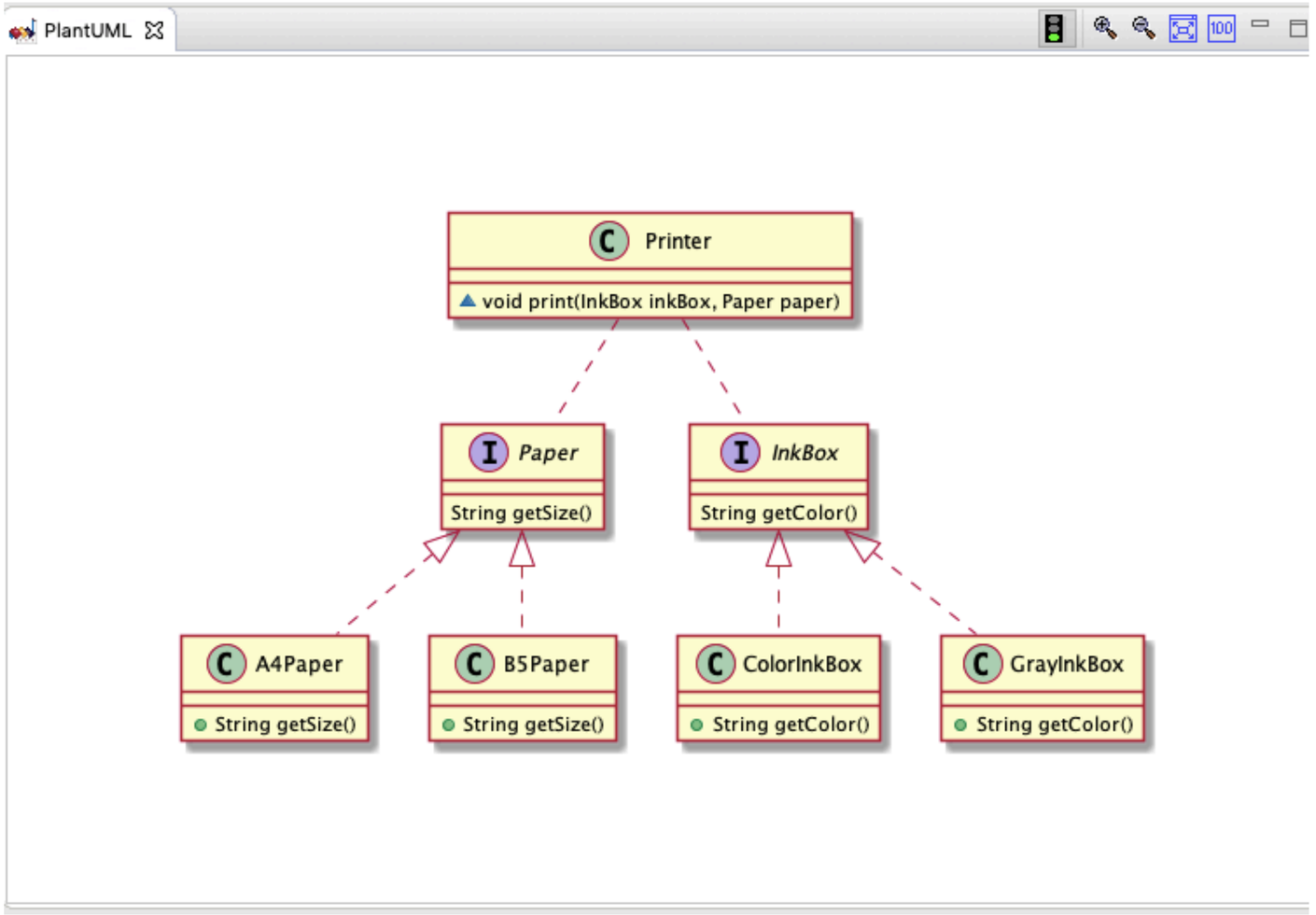
  - 程序设计时面向接口的约定而不考虑具体实现

# 面向接口编程

- 开发打印机
  - 墨盒：彩色、黑白
  - 纸张类型：A4、B5
  - 墨盒和纸张都不是打印机厂商提供的
  - 打印机厂商要兼容市场上的墨盒、纸张

# 类图

```
@startuml
class Printer {
        ~void print(InkBox inkBox, Paper paper)
}

class A4Paper {
        +String getSize()
}
interface Paper {
}
Paper <|.. A4Paper

class B5Paper {
        +String getSize()
}
interface Paper {
}
Paper <|.. B5Paper

class ColorInkBox {
        +String getColor()
}
interface InkBox {
}
InkBox <|.. ColorInkBox

class GrayInkBox {
        +String getColor()
}
interface InkBox {
}
InkBox <|.. GrayInkBox

interface InkBox {
        String getColor()
}

interface Paper {
        String getSize()
}
Printer ..Paper
Printer ..InkBox

@enduml
```



# 接口表示一种能力

- 接口是一种能力

  体现在接口的方法上

- 面向接口编程

  程序设计时
  - 关心实现类有何能力，而不关心实现细节
  - 面向接口的约定而不考虑接口的具体实现

# 静态内部类

- 不能访问外部类的非静态成员
- 在外面访问此类的话就用 外部类名.内部类名
- 比如

```
OuterA.InnerA inn=new OuterA.InnerA();
```

- 创建了一个静态内部类对象
- 静态内部类和外部类对象实际上是没什么关系的

# 静态内部类

- 补充：
  - 静态内部类：作为一个类成员存在
    - 修饰符和成员内部类相同public protected (Default) private abstract final static
    - 不能访问外部类的非静态成员，可以访问静态成员
    - 实例化(new 外部类名.内部类构造函数())
    - 声明(外部类名.内部类名 变数)
    - 静态内部类里可以定义静态的属性和方法

# 成员内部类

- 可以访问外部类的静态和非静态成员
  - 访问成员内部类必须先生成一个外部类对象，生成局部内部类的对象用

```
外部类名.内部类名 对象名=外部类对象.new 内部类名();
```

  - 比如

```
OuterA out=new OuterA();
OuterA.InnerB innb=out.new InnerB();
```

- 如何选择用静态内部类或成员内部类：
  - 如果外部类怎么变化都不影响内部类的话，那么推荐用静态内部类，
  - 如果操作内部类的改变能影响外部类，那么推荐使用成员内部类。

# 成员内部类

- 补充：
  - 成员内部类：作为一个对象(实例)的成员存在
    - 修饰符可以是public protected (Default) private abstract final static
    - 可以访问外部类的成员(外部类名.this.成员)
    - 实例化(外部类对象的引用.new 内部类构造函数())
    - 声明(外部类名.内部类名 变数)
    - 成员内部类里不能定义静态的属性和方法

# 局部内部类

- 在方法体里定义的类
- 如何在方法体外得到这个类的对象呢，通过方法的返回值就可以，返回值类型为Object。
- 但是外部得到的这个对象是没什么意义的，因为只能访问Object里的方法，此类自己的方法是访问不到的。
- 解决方法：
  - 1、覆盖Object类的方法，此方法不推荐。
  - 2、在外部定义一个接口，然后此局部内部类来实现接口，然后外部拿到对象后转化成此接口类型就行了。
  - 3、反射技术

# 局部内部类

- 局部内部类中不能访问外部方法中的局部变量，如果要访问的话，局部变量必须是final。
- 因为对象是存在于堆中的，而方法是存在于栈中的，对象对方法中局部变量访问的话就要创建局部变数的一个副本来存放此局部变量的值，所以其中有一个改变了，另外一个是不会改变的，所以为了保持一致性，要把这个局部变量定义为final。
- 局部内部类中不能生成静态成员
- 局部内部类：定义在方法体里的类
- 修饰符可以是abstract final
- 可以访问外部类的成员
- 声明和实例化跟普通类一样
- 局部内部类里不能定义静态的属性和方法
- 局部内部类只能访问final修饰的局部变数

# 匿名内部类（重点掌握）

- 没有名字的类
- 匿名内部类的声明和对象的生成是合二为一，不可分割的
- 得到的对象不知道是哪个类的对象，只知道是实现了某接口或者继承了某类的类的对象
- 匿名内部类没有构造函数（因为没有名字），也不能被继承
- 匿名内部类可以访问外部类的成员：外部类名.this.成员
- 匿名内部类如果声明在方法中的话也具有局部内部类的特点，只能调用final的局部变量
- 匿名内部类中不能声明静态的成员

匿名内部类的声明

```
声明类型 变量=new [类名 extends/implements] 声明类型(){
        类的方法声明...
    }
    []中的是省略的部分
```

# 匿名内部类（重点掌握）

- 补充：
  - 匿名内部类：没有类名的类
  - 可以访问外部类的成员
  - 声明和实例化一起(声明类型 变量=new 声明类型(){类的定义})；
  - 声明类型：抽象类 接口 普通类
  - 匿名内部类如果定义在方法体里，则具有和局部内部类相同的特征
  - 匿名内部类可以定义在属性的赋值上

# THE END