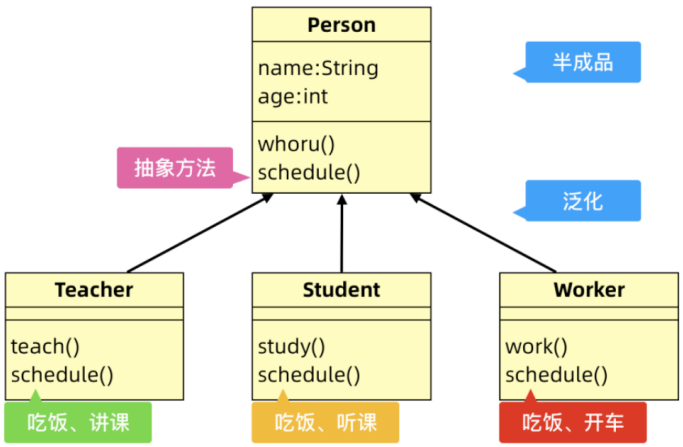
面向对象 DAY05 晚课

1. 串讲内容

1. 抽象方法和抽象类

1.1 抽象方法

- **定义：**使用 abstract 关键字声明，不包含方法体的方法称之为抽象方法
- **原因：**在利用泛化设计父类的时候，将全体子类都共有的方法抽取到父类中，但是每个具体的方法实现都不相同，这样只能在父类中声明方法的定义，在子类中去实现，因此在父类中的方法就是一个不完整的方法，需要定义为抽象方法。

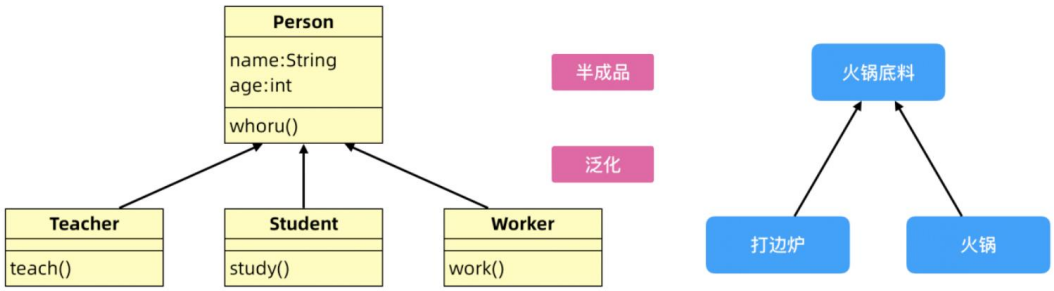


- **抽取规则：**
  - 将每个子类都有，但是每个子类实现都不同的方法泛化为抽象方法
  - 将每个子类都有，并且实现都一样的方法泛化到父类中
- **语法规则：**
  - 使用 abstract 声明方法，不能有方法体
  - 包含抽象方法的类必须声明为抽象类，因为包含抽象方法的类一定是不完整的半成品类
  - 子类继承抽象类时必须重写（实现）抽象方法，否则出现编译错误
    - 可以将抽象方法看做时父类对子类的行为约定，必须被子类重写实现

- **好处：**
  - 使用抽象方法的好处是：可以将抽象方法看作时父类对子类声明的行为约定，必须由子类遵守实现，由于 Java 编译器的语法检查子类是否实现了方法，这样则可以避免开发者实现的时候意外落下没有重写的方法。

1.2 抽象类

- **定义：**使用抽象关键字 abstract 声明的类是抽象类，抽象类不能直接实例化创建对象。
- **原因：**在面向对象设计的时候，会利用”泛化“将子类的共同属性和方法抽取出来设计出父类，此时的父类往往是半成品类，只包含部分属性和方法，甚至属性值都没有合理初始化，如果直接创建对象并且使用会造成各种不理想的结果，甚至是异常故障。



- **定义规则：**面向对象设计时根据子类泛化得到的半成品父类，应该定义为抽象类，这样可以限制创建半成品类的对象，减少意外的错误发生
- **如何使用：**
  - 在类名前面添加 abstract 关键字
  - 抽象类可以作为父类被子类继承，可以定义变量，可以定义抽象方法，也可以定义非抽象方法
  - 抽象类不能直接创建对象，需要被子类继承
- **抽象类的意义：**
  - 封装子类共有的属性和行为----以便达到代码复用
  - 可以包含抽象方法，为所有子类提供统一的入口（方法名统一），并要求子类必须强制重写

2. 接口

2.1 接口的语法

- 接口是一种引用数据类型，可以定义引用类型变量，由 interface 声明
- 接口中只能定义常量和抽象方法
  - 可以省略常量的修饰词 public static final
  - 可以省略抽象方法修饰符 public abstract
  - 在 JDK1.8 之后接口中可以定义非抽象的有方法实现的方法，但是需要使用 default 关键字声明

```java
public interface Inter {
     void  print();
    default  void  test(){} //默认方法
    static  void print(){} //静态方法
}
```

- 接口不能实例化创建对象，可以作为父类型被子类实现（继承）
- 接口用于定义多个类的共同行为和规范，需要被子类实现，通过 implements 关键字实现接口
- 一个类可以实现多个接口，用逗号分隔
- 子类实现接口时必须重写接口中的全部抽象方法

2.2 抽取规则

在 Java 编程语言中接口是一个抽象类型，这样就会带来一个矛盾，抽象类也是抽象的，接口也是抽象的，区别在哪里？

- **统一种类的公共行为和属性可以抽取到抽象类**
  - **如果所有子类的行为都一样，设计为普通方法**
  - **如果所有子类的行为不一样，设计为抽象方法**
- **不同种类的公共行为抽取到接口中**
  - **接口是对继承的单根性的扩展----实现多继承**
  - **符合既是也是的原则**
- 例如：
  - 鸟 Bird
    - 飞行方法，觅食方法

- 狗 Dog
    - 爬行方法，觅食方法
- 老虎 Tiger
    - 跑方法，觅食方法
- 青蛙 Frog
    - 跳方法，觅食方法
- 以上子类都有觅食方法，根据 java 继承的抽取归纳原则，将所有子类所共有的方法抽取到父类，又由于每个子类的实现不同，所以将父类中的方法定义为抽象方法
- 父类 Animal 类

```java
public abstract class Animal{
    public abstract void food();
}
```

- 而飞行这些都是部分子类所具备的行为，所以定义为接口，让拥有这个能力的子类去实现

```java
interface Swim{
    void swim();
}
interface crawl{
    void crawl();
}
```

2.3 接口的多继承

在生活中是存在多继承现象的：

- 银币：即是银子，也是钱
- 纸币：即是纸张，也是钱

Java 不支持“类的多继承”，也就是一个 Java 类不能继承多个父类，但是 Java 支持一个类继承一个父类实现多个接口，Java 利用这种方式实现了多继承：Java 利用接口实现了多继承

一个接口可以同时继承多个接口，中间用逗号隔开

```java
public interface Inter extends Inter1,Inter2{
```

```
    default  void  test(){} //默认方法
    static  void print(){} //静态方法
}

interface Inter1{}

interface  Inter2{}
```

2.4 接口的多实现

接口不允许实例化对象，接口需要被类实现，一个类可以同时实现多个接口

- 注意：
  - 在一个类既有继承，又有实现的时候，是先继承后实现
  - 类和类之间是继承关系----单继承
  - 接口和接口之间也是继承关系----多继承
  - 类和接口之间是实现关系----多实现
- 例如：
  - 定义动物抽象类，声明所有动物所具备的行为，例如觅食行为
  - 爬行和游泳是部分动物所具备的行为，分别定义两个接口，让具体此行为的动物来实现
  - 定义子类青蛙类
- Animal 类

```
Java
abstract class Animal{
    public abstract void food(); //所有子类共有的
}
```

- 爬行接口 Crawl，游泳接口 Swim

```
Java
interface Swim{  //游泳接口
    void swim();
}
interface crawl{  //爬行接口
```

```
    void crawl();
}
```

- 青蛙类 frog,青蛙即是动物又会爬行还会游泳，所以需要继承动物类，并实现游泳接口和爬行接口

```
Java
class Frog extends Animal implements Swim,crawl{

    @Override
    public void food() {
        System.out.println("青蛙吃虫子");
    }

    @Override
    public void swim() {
        System.out.println("青蛙会游泳");
    }

    @Override
    public void crawl() {
        System.out.println("青蛙会爬行");
    }
}
```

- 当既有继承又有实现的时候，类先继承一个类，再实现接口

3. 多态

多态：多种表现形态

多态有两种表现形式：

- 行为（方法）多态：同一个方法，在不同的对象身上有不同的表现
- 对象多态：同一个对象，被造型成不同的形态时，所具备的功能不同

3.1 向上造型

- **定义**

Java 中可以将子类型对象赋值给父类型变量，这种现象成为向上造型。 向上造型的好处是父类型变量可以引用各种子类型的实例，这样就可以实现多态。

父类引用指向了子类对象

- **优点**

父类型变量可以引用各种子类型的实例，这样就可以实现多态。

其实不管是行为多态也好，还是对象多态也罢，他们都是在造型，所以多态一定免不了向上造型，如果只有一种形态的话，是没有多态的。

- 能造型成的类型：父类+所实现的接口
- 例如：定义一个 Animal 父类，飞行 Flight 接口，以及 Crawl 接口

```
Java
abstract class Animal{   //父类
    public abstract void eat();
}
interface Crawl{    // 爬行接口
    void crawl();
}
interface Flight{   //飞行接口
    void flight();
}
```

- 定义三个子类，分别继承父类，实现对应的接口

```
Java
//小猫类继承动物类实现爬行接口
class Cat extends Animal implements Crawl{
    @Override
    public void eat() {
        System.out.println("小猫吃猫粮");
    }

    @Override
    public void crawl() {
        System.out.println("小猫爬行");
    }
}
//青蛙类继承动物类实现爬行接口
class Frog extends  Animal implements Crawl{
    @Override
    public void crawl() {
        System.out.println("青蛙会爬");
    }
```

```
    @Override
    public void eat() {
        System.out.println("青蛙吃蚊子");
    }
}
//小鸟类继承动物类实现飞行接口
class Bird extends  Animal implements Flight{
    @Override
    public void flight() {
        System.out.println("小鸟会飞");
    }
    @Override
    public void eat() {
        System.out.println("小鸟吃虫子");
    }
}
```

- 测试

```
Java
/**
 * 方法多态：
 * 同一个类型，作用于不同对象的时候，调用同一个方法，所具备的实现不同。
 * 通过向上造型实现：所能造型的类型为 父类+所实现的接口
 * 当向上造型的时候，所能点出哪些方法，看声明的类型中有哪些方法
 */
Animal animal;  //声明父类类型
animal=new Cat(); //将父类类型造型为小猫对象
animal.eat(); //小猫吃猫粮


animal=new Frog(); //将父类类型造型为青蛙对象
animal.eat(); //青蛙吃蚊子

animal=new Bird(); //将父类类型造型为小鸟对象
animal.eat(); //小鸟吃虫子
```

```
Crawl crawl; //声明爬行接口
crawl = new Cat();
crawl.crawl(); //小猫爬行
crawl = new Frog();
crawl.crawl(); //青蛙会爬
```

3.2 向下转型

- **定义**

与向上造型相反，将父类型引用的对象赋值给子类型变量

- **使用场景**

父类型的引用只能调用父类型的方法,如果调用子类型方法不可以,通过向下转型可以解决这一问题

- **弊端**

“向下转型”存在造型异常风险， 因为父类型引用的对象有可能不是具体子类型的对象

- 做向下转型的时候，引用类型转换的话可能会有异常，成功的条件只有如下两种
  - 引用所指向的对象，就是该类型
  - 引用所指向的对象，实现了该接口
- 案例：父类和接口同上

```
Java
/**
 * 向下转型：将父类型引用的对象赋值给子类型变量
 * 当向上造型时，能调用哪些方法看的是引用的类型，就无法调用全部的子类方法
 * 可以通过向下转型来实现
 *
 * 弊端：向下转型存在风险，所造型的对象并不是同一类型
 */
Cat cat=(Cat) animal;
cat.crawl();
cat.eat();
cat.play();
```

4. 内部类

- 成员内部类：应用率低，了解
  - 类中套类，外面的称为外部类，里面的称为内部类
  - 内部类通常只服务于外部类，对外不具备可见性
  - 内部类对象通常在外部类中创建
  - 内部类可以直接访问外部类的成员，在内部类中有个隐式的引用指向创建它的外部类对象
- *隐式的引用：外部类名.this*
  - 何时用：若 A 类(Baby)只让 B 类(Mama)用，并且 A 类(Baby)还想访问 B 类(Mama)的成员时，可以设计成员内部类
- 匿名内部类：应用率高，掌握
  - 何时用：若想创建一个派生类的对象，并且对象只创建一次，可以设计为匿名内部类，可以大大简化代码
  - 注意：匿名内部类中不能修改外面局部变量的值
  - 小面试题：
    - 问：内部类有独立的.class 吗?
    - 答：有
- final：最终的、不能改变的----------单独应用几率低
  - 修饰变量：变量不能被改变
  - 修饰方法：方法不能被重写
  - 修饰类：类不能被继承
- static：静态的
  - 静态变量：
    - 由 static 修饰
    - 属于类，存储在方法区中，只有一份
    - 常常通过类名点来访问
    - 何时用：对象所共享的数据
  - 静态块：
    - 由 static 修饰

        - 属于类，在类被加载期间自动执行，一个类只被加载一次，所以静态块也只执行一次
        - 何时用：初始化/加载静态资源/静态变量
    - 静态方法：
        - 由 static 修饰
        - 属于类，存储在方法区中，只有一份
        - 常常通过类名点来访问
        - 静态方法中没有隐式 this 传递，所以静态方法中不能直接访问实例成员(实例变量/实例方法)
        - 何时用：方法的操作与对象无关(不需要访问对象的属性/行为)
- static final 常量：应用率高
    - 必须声明同时初始化
    - 常常通过类名点来访问，不能被改变
    - 建议：常量名所有字母都大写，多个单词之间用_分隔
    - 编译器在编译时，会将常量直接替换为具体的数，效率高
    - 何时用：在程序运行过程中数据永远不变，并且经常使用
- 枚举：
    - 是一种引用数据类型
    - 特点：枚举类型的对象数目是固定的，常常用于定义一组常量
- *例如：季节、星期、月份、订单状态、性别.....*
    - 所有枚举都继承自 Enum 类，其中提供了一组方法供我们使用
    - 枚举的构造方法都是私有的

2. 面试题

2.1 接口和抽象类的区别

接口（Interface）和抽象类（Abstract Class）是 Java 中两种不同的机制，用于实现面向对象编程中的抽象和多态性。它们在定义和使用上有以下区别：

1. **定义方式：**

- 接口：使用 `interface` 关键字定义，可以包含常量和抽象方法，不包含具体实现的方法。
- 抽象类：使用 `abstract` 关键字修饰的类，可以包含抽象方法和具体实现的方法，也可以包含字段和构造方法。

1. **多继承限制：**

- 接口：一个类可以实现多个接口，实现了接口的类可以同时具备多个接口的特性。
- 抽象类：一个类只能继承一个抽象类，因为 Java 不支持多继承，但可以通过实现多个接口来获得类似的效果。

1. **方法实现：**

- 接口：接口中的方法都是抽象方法，没有方法体，需要由实现接口的类提供具体实现。
- 抽象类：抽象类中可以包含抽象方法和具体实现的方法，子类可以直接继承和使用具体实现的方法，也可以选择重写抽象方法。

1. **构造方法：**

- 接口：接口不能有构造方法，因为接口主要是用于定义规范，不涉及具体的实例化过程。
- 抽象类：抽象类可以有构造方法，用于初始化抽象类的属性和调用父类的构造方法。

1. **成员变量：**

- 接口：接口中只能定义常量，而且常量默认为 `public static final` 类型。
- 抽象类：抽象类可以包含普通字段和常量，字段的访问修饰符可以根据需要进行定义。

1. **使用场景：**

- 接口：适用于定义多个类的共同行为和规范，实现类通过实现接口来达到多态性的目的。
- 抽象类：适用于定义一组相关的类的基本行为和属性，子类通过继承抽象类来获取共同的特性。

总结：
接口主要用于定义规范和多态性，强调行为的一致性；抽象类主要用于封装共同行为和属性，强调类的层次结构。在使用上，接口适合用于设计多个具有不同特征的类，而抽象类适合用于构建具有相似特征的类的继承关系。

2.2 实例变量和静态变量的区别

实例变量（Instance Variables）和静态变量（Static Variables）是 Java 中两种不同类型的变量，它们在定义、作用域和内存分配等方面有以下区别：

在类中定义的数据

```Java
class A{
  int a; //实例变量
  static int b ;  //静态变量
  static final int MAX； //静态常量
}
```

1. **定义：**

- 实例变量：实例变量是定义在类中，但在方法、构造方法和代码块之外的变量。每个类的实例（对象）都有自己的一组实例变量，它们的值在每个对象中可以是不同的。
- 静态变量：静态变量是用 `static` 关键字声明的变量，它属于类本身而不是类的实例。所有该类的对象共享相同的静态变量，即它们具有相同的值。

1. **作用域：**

- 实例变量：实例变量的作用域在对象级别，即每个对象都有自己的一组实例变量。可以通过对象访问实例变量。
- 静态变量：静态变量的作用域在类级别，它属于整个类而不是类的任何特定实例。可以通过类名直接访问静态变量。

1. **内存分配：**

- 实例变量：每个对象都有自己的实例变量，它们在对象创建时分配内存，在对象被销毁时释放内存。
- 静态变量：静态变量在类加载时分配内存，并在整个程序运行期间保持不变，直到程序结束或显式修改其值。

1. **访问方式：**

- 实例变量：实例变量需要通过对象访问，即使用对象名来访问。
- 静态变量：静态变量可以直接通过类名访问，也可以通过对象访问，但推荐使用类名来访问。

1. **初始化时机：**

- 实例变量：实例变量可以在定义时初始化，也可以在构造方法中初始化。
- 静态变量：静态变量可以在定义时初始化，也可以在静态代码块中初始化。

总结：
实例变量是每个对象特有的变量，具有对象级别的作用域和内存分配，需要通过对象访问；静态变量属于类本身，具有类级别的作用域和内存分配，可以通过类名直接访问。实例变量适用于描述对象的特征和状态，静态变量适用于描述类的特征和共享数据。

2.3 java 参数传递是值传递还是引用传递

在 Java 中无论是基本类型还是引用类型，参数传递都是值传递

```
Plain Text
class A{
 public void  a(int a){
 }
 public void b(Student s){
 }
 public void c(){
   a(3);
   b(new Student())
 }
}
```

- 在值传递中，方法调用时，实际参数的值被复制给对应的形式参数，方法内部操作的是这个副本，而不是原始参数本身，这意味着，在方法内部修改形式参数的值不会影响原始参数的值
- 当将一个引用类型作为参数传递给方法时，实际上传递的是该引用的副本（也就是引用的值），而不是引用所指向的对象本身，因此，方法内部对于引用类型参数的修改只影响了副本的值，而不会改变原始引用的指向。
- 总结：
  - Java 中采用值传递，即方法调用时，实参的值被赋值给形参，方法内部操作的是这个副本，对于引用类型的变量，副本是引用的值，因此可以通过引用来访问和修改对象的状态，但是无论如何，方法内部的修改都不会影响原始参数的值，这是因为方法内部操作的是副本，而不是原始参数本身。

2.4 为什么构造方法不能被继承

构造方法不能被继承是因为构造方法的作用是用于创建对象并初始化对象的状态，它在对象创建过程中起到了特殊的作用。构造方法具有以下特点：

1. 构造方法的名称必须与类名相同，并且没有返回类型。
2. 构造方法在对象创建时自动调用，用于初始化对象的实例变量和执行其他必要的操作。
3. 构造方法可以有多个重载形式，根据参数的不同进行重载。

当子类继承父类时，子类会继承父类的属性和方法，但构造方法并不会被继承。这是因为构造方法具有以下特殊性：

1. 构造方法的作用是创建对象并初始化其状态，这个过程是针对特定类的，子类的构造方法用于初始化子类自己的实例变量，而不是父类的实例变量。

2. 父类的构造方法中可能包含特定于父类的逻辑和操作，这些逻辑在子类中可能不适用或者需要不同的处理方式。

如果构造方法可以被继承，那么子类就会自动继承父类的构造方法，这可能导致以下问题：

1. 子类对象的创建和初始化可能无法满足子类特定的需求，因为子类可能有自己独特的属性需要初始化。

2. 父类构造方法中的逻辑和操作可能不适用于子类，导致不一致的行为。

因此，为了确保对象的正确初始化和保持逻辑的一致性，构造方法不能被继承，子类需要自己定义并实现自己的构造方法，以满足子类的特定需求。