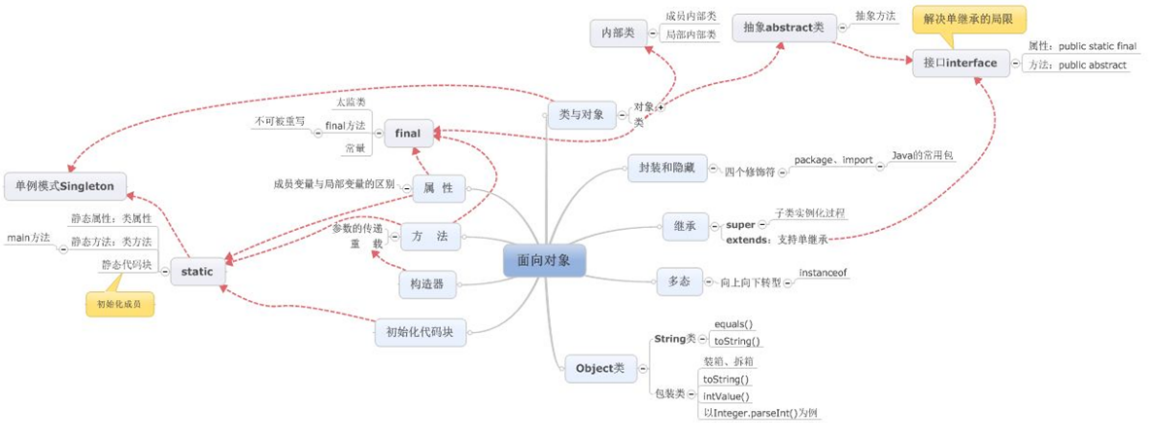
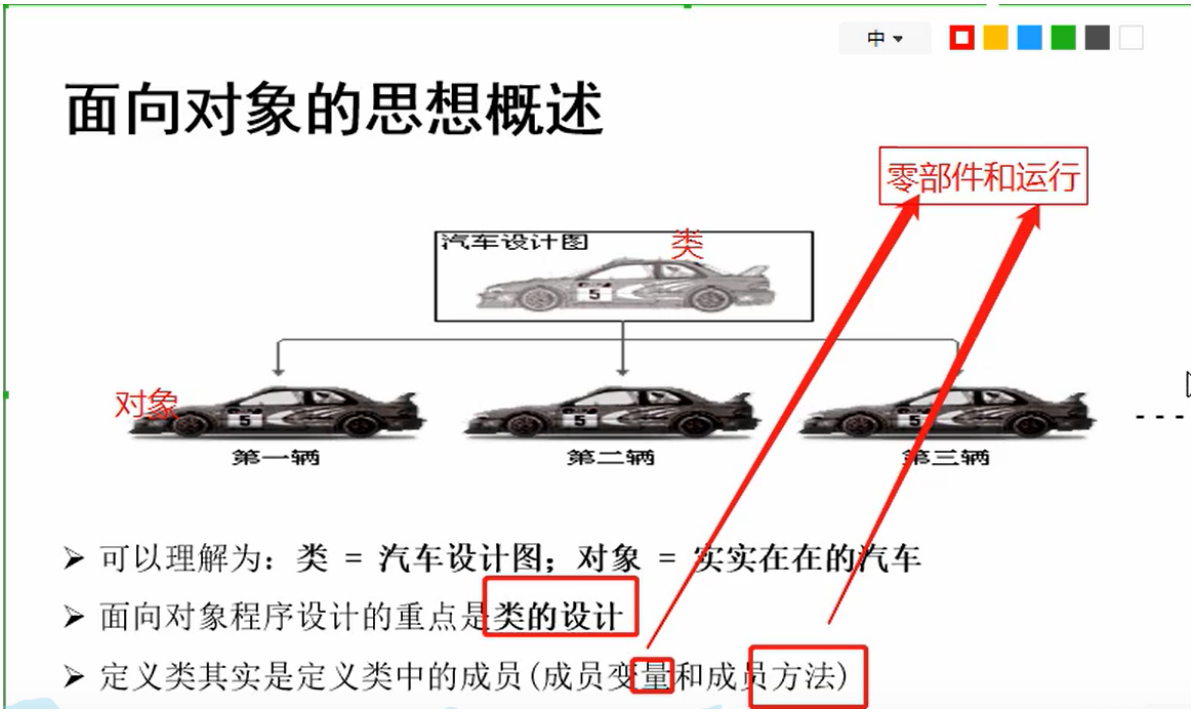
面向对象编程



1 概述

类的三大成员: 属性,方法,构造器(new 对象实际就是调用类的构造方法.)

类的三大特性:封装,继承,和多态



1.1 可变参数的传递

- **数组形式**: `public static void test(int a ,String[] books);`
- **… 形式:** `public static void test(int a ,String…books);`
- **二者区别在于数组传递不可为空,… 可为空传递**

1.2 java传递全部为值传递

java 虚拟机有三块内存区域:分别为 **堆区, 栈区, 代码区.** 一般基本数据类型和引用变量的内存存在于栈区,而引用对象实体存在于堆区. java的函数传递方式为值传递,若将 `i = 3` 传入其他函数,是将3这个值给了函数局部变量,但若将引用对象传入函数 `Person p1 = New Person()`,是将person实体的地址传入函数了.

- **方法的参数传递**
  1、如果方法的形参是基本数据类型，那么实参（实际的数据）向形参传递参数时，就是直接传递值，把实参的值复制给形参。
  2、如果方法的形参是对象，那么实参（实际的对象），向形参传递参数的时，也是把值给形参，这个值是实参在栈内存中的值，也就是引用对象在堆内存中的地址
  基本数据类型都是保存在栈内存中，引用对象在栈内存中保存的是引用对象的地址，那么方法的参数传递是传递值（是变量在栈内存的当中的值

1.3 构造器

- 使用为类方法的同名函数,可用于给对象初始化值
- 一个类可以创建多个重载的构造器
- 默认构造器的修饰符与所属类的修饰符一致,父类的构造器不可被子类继承

1.4 this 关键字

this表示当前对象，可以调用类的属性、方法和构造器,使用场景如下:

1. 当形参与成员变量重名时，如果在方法内部需要使用成员变量，必须添加this来表明该变量时类成员
2. 在任意方法内，如果使用当前类的成员变量或成员方法可以在其前面添加this，增强程序的阅读性
3. this可以作为一个类中，构造器相互调用的特殊格式

2 封装

2.1 封装概念

高内聚,低耦合.

Java中通过将数据声明为私有的(private)，再提供公共的（public）方法:getXxx()和setXxx()实现对该属性的操作，以实现下述目的：

- 隐藏一个类中不需要对外提供的实现细节；
- 使用者只能通过事先定制好的方法来访问数据，可以方便地加入控制逻辑，限制对属性的不合理操作；
- 便于修改，增强代码的可维护性；

2.2 四种接口访问权限

修饰符	类内部	同一个包	子类	任何地方
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

3 继承

3.1 继承概念

`class Subclass extends Superclass{ }`多个类中存在相同属性和行为时，将这些内容抽取到单独一个类中，那么多个类无需再定义这些属性和行为，只要继承那个类即可。

作用：

- 继承的出现提高了代码的复用性。
- 继承的出现让类与类之间产生了关系，提供了多态的前提。
- 不要仅为了获取其他类中某个功能而去继承

特点:

- `Java`只支持单继承，不允许多重继承(一个父亲可以有多个孩子,一个孩子不能够有多个父亲)
- 方法重写 != 方法重载,重写快捷键为 `Ctrl+O`

3.2 super 关键字

在Java类中使用super来调用父类中的指定操作

3.2.1 作用:

- 尤其当子父类出现同名成员时，可以用super进行区分
- super的追溯不仅限于直接父类
- super和this的用法相像，this代表本类对象的引用，super代表父类的内存空间的标识

3.3.2 调用父类的构造器:

子类中所有的构造器**默认**都会访问父类中**空参数**的构造器

当父类中没有空参数的构造器时，子类的构造器必须通过**this(**参数列表**)**或者**super(**参数列表)语句指定调用本类或者父类中相应的构造器，且必须放在构造器的第一行

l如果子类构造器中既未显式调用父类或本类的构造器，且父类中又没有无参的构造器，则**编译出错**

3.2.3 与this区别:

No.	区别点	this	super
1	访问属性	访问本类中的属性，如果本类没有此属性则从父类中继续查找	访问父类中的属性
2	调用方法	访问本类中的方法	直接访问父类中的方法
3	调用构造器	调用本类构造器，必须放在构造器的首行	调用父类构造器，必须放在子类构造器的首行
4	特殊	表示当前对象	无此概念

4 多态

Java引用变量有两个类型：**编译时类型**和**运行时类型。**编译时类型由声明该变量时使用的类型决定，运行时类型由实际赋给该变量的对象决定。**若编译时类型和运行时类型不一致，就出现多态**（Polymorphism）

前提:

- 需要存在继承或者实现关系
- 要有覆盖操作

成员方法:

- 编译时：要查看引用变量所属的类中是否有所调用的方法。
- 运行时：调用实际对象所属的类中的重写方法。

成员变量:

- 不具备多态性，只看引用变量所属的类。

4.1 instance of 操作符 和 object 类

- x instanceof A：检验x是否为类A的对象，返回值为boolean型。
- object 类的主要方法:

NO.	方法名称	类型	描述
1	public Object()	构造	构造方法
2	public boolean equals(Object obj)	普通	对象比较
3	public int hashCode()	普通	取得Hash码
4	public String toString()	普通	对象打印时调用

5 对象类型及其他重要问题

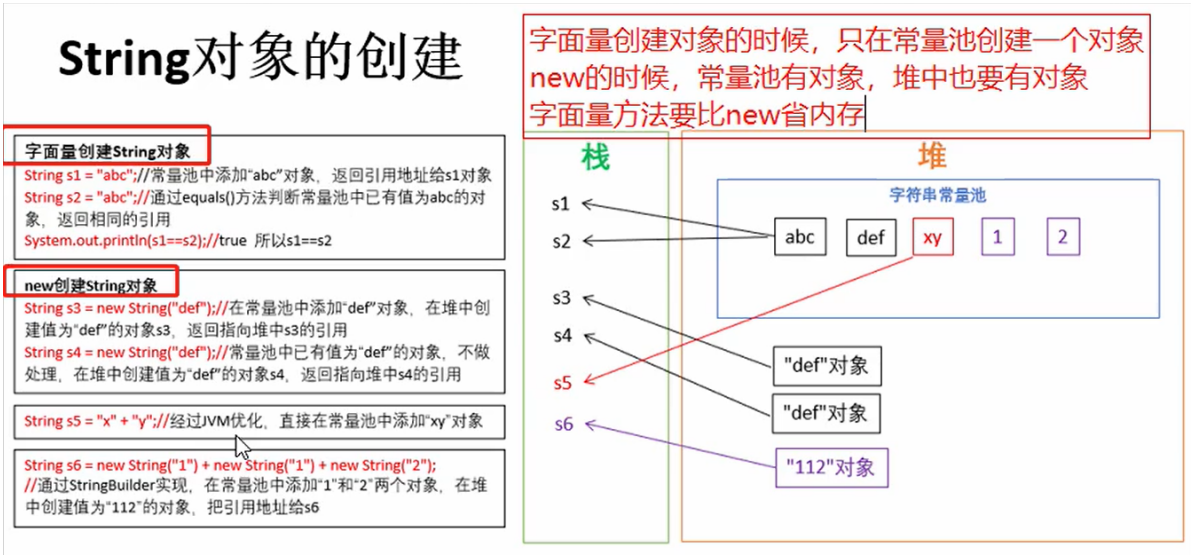
5.1 类型转换

- **自动类型转换**：小的数据类型可以自动转换成大的数据类型　如long g=20;　　double d=12.0f
- **强制类型转换**：可以把大的数据类型强制转换(casting)成小的数据类 如 float f=(float)12.0；int a=(int)1200L

5.2 "equals" 与 "=="

1. 对于普通数据类型: 只能用 == 来比较两个对象的**值**是否相等
2. 对于引用数据类型:
    - 一般情况下比较二者的引用内存是否一致
    - 当用equals()方法进行比较时，对类File、String、Date及包装类（Wrapper Class）来说，是比较类型及内容而不考虑引用的是否是同一个对象；
      原因：在这些类中重写了Object类的equals()方法。

5.3 String 类




```
        String s1 = new String("ghgh");
        String s2 = new String("ghgh");
        String s3 ="ghgh";
        String s4 ="ghgh";
        System.out.println(s1.equals(s2));
        System.out.println(s1 == s2);
        System.out.println(s3.equals(s4));
        System.out.println(s3 == s4);
```

依次输出结果为 `true  false  true  true`

另:

- toString()方法在Object类中定义，其返回值是String类型，返回类名和它的引用地址。
- 包装类的作用就是让其他数据类型和String之间进行数据类型转换

5.4 单例设计模式

如果一个对象在`new`时候需要大量时间和内存资源,采取某些方法让他只创建一次对象,后面的直接共享内存.有懒汉式和饿汉式两种模式.

1. **懒汉式**:

```
class Single{
    //private的构造器，不能在类的外部创建该类的对象
     private Single() {}
     //私有的，只能在类的内部访问
     private static Single onlyone = new Single();
    //getSingle()为static，不用创建对象即可访问
     public static Single getSingle() {
        return onlyone;
    }
}
 public class TestSingle{
    public static void main(String args[]) {
        Single  s1 = Single.getSingle();      //访问静态方法
        Single  s2 = Single.getSingle();
        if (s1==s2){
            System.out.println("s1 is equals to s2!");
        }}}
```

2.饿汉式:

```
class Singleton{
    //1.将构造器私有化，保证在此类的外部，不能调用本类的构造器。
    private Singleton(){
    }
    //2.先声明类的引用
    //4.也需要配合static的方法，用static修饰此类的引用。
    private static Singleton  instance = null;
    //3.设置公共的方法来访问类的实例
    public static Singleton  getInstance(){
    //3.1如果类的实例未创建，那些先要创建，然后返回给调用者：本类。因此，需要static 修饰。
        if(instance == null){
            instance = new Singleton();
        }
    //3.2 若有了类的实例，直接返回给调用者。
        return instance;
    }  }
```

5.5 初始化快

> 在实际开发中， static静态代码块用在初始化类的静态属性（ static类型属性）
>
> 匿名内部类上代码块会发挥作用(只能用代码块初始化)

6 抽象类,接口与工厂模式

6.1 抽象类与模板设计方法

抽象类:

- 用abstract关键字来修饰一个类时，这个类叫做抽象类；
- 用abstract来修饰一个方法时，该方法叫做抽象方法。

> 抽象方法：只有方法的声明，没有方法的实现。以分号结束：abstract int abstractMethod(int a);

- 含有抽象方法的类必须被声明为抽象类。
- 抽象类不能被实例化。抽象类是用来作为父类被继承的，抽象类的子类必须重写父类的抽象方法，并提供方法体。若没有重写全部的抽象方法，仍为抽象类。
- 不能用abstract修饰属性、私有方法、构造器、静态方法、final的方法。

模板设计方法:

- 当功能内部一部分实现是确定，一部分实现是不确定的。这时可以把不确定的部分暴露出去，让子类去实现。
- 编写一个抽象父类，父类提供了多个子类的通用方法，并把一个或多个方法留给其子类实现，就是一种模板模式。

eg:

```
abstract class Template{
    public final void getTime(){
        long start = System.currentTimeMillis();
        code();
        long end = System.currentTimeMillis();
        System.out.println("执行时间是："+(end - start));
    }
    public abstract void code();
}
class SubTemplate extends Template{
    public void code(){
        for(int i = 0;i<10000;i++){
        System.out.println(i);
} } }
```

6.2 接口

有时必须从几个类中派生出一个子类，继承它们所有的属性和方法。但是，Java不支持多重继承。有了接口，就可以得到多重继承的效果.

```
interface Runner { public void run();}
interface Swimmer {public double swim();}
class Creator{public int eat(){…}}
class Man extends Creator implements Runner ,Swimmer{
        public void run() {……}
        public double swim()  {……}
        public int eat() {……}
```

特点:

- 接口中的所有成员变量都默认是由public static final修饰的。

- 接口中的所有方法都默认是由public abstract修饰的。
- 如果实现接口的类中没有实现接口中的全部方法，必须将此类定义为抽象类
- 实现接口的类中必须提供接口中所有方法的具体实现内容，方可实例化。否则，仍为抽象类。
- 接口的主要用途就是被实现类实现。

6.3 抽象类与接口的区别

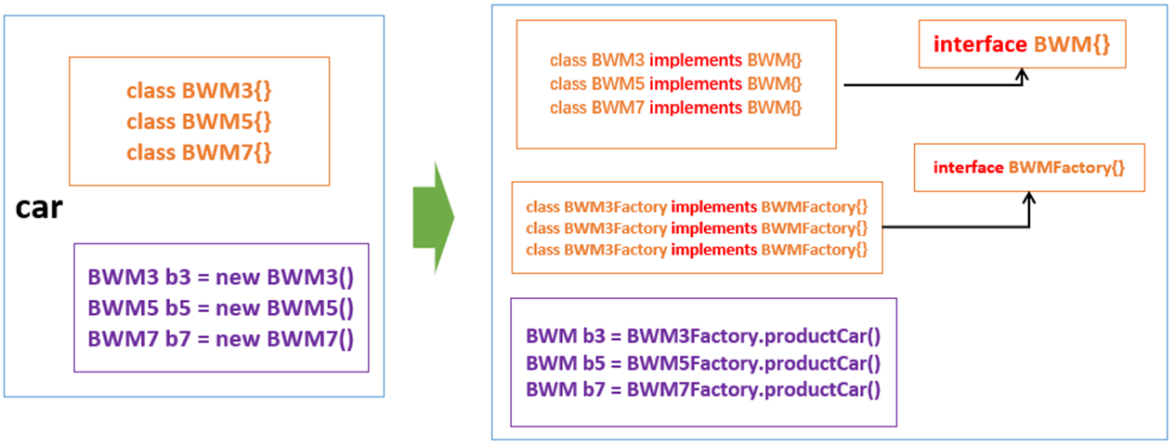
抽象类是对于一类事物的高度抽象，其中既有属性也有方法
接口是对方法的抽象，也就对系列动作的抽象
当需要对一类事物抽象的时候，应该是使用抽象类，好形成一父类
当需要对一系列的动作抽象，就使用接口，需要使用这些动作的类去实现相应的接口即可

6.4 工厂设计模式

FactoryMethod模式是设计模式中应用最为广泛的模式，在面向对象的编程中，对象的创建工作非常简单，对象的创建时机却很重要。FactoryMethod解决的就是这个问题，它通过面向对象的手法，将所要创建的具体对象的创建工作延迟到了子类，从而提供了一种扩展的策略，较好的解决了这种紧耦合的关系。

通过工厂把new对象给隔离，通过产品的接口可以接受不同实际产品的实现类，实例的类名的改变不影响其他合作开发人员的编程.

`eg:`



6.5 内部类

> 内部类的最大作用是 **实现多重继承**

7 总结