封装：

|  |  |
| --- | --- |
| 定义 | 将对象的状态信息一些隐藏在对象内部，不允许外部程序直接访问对象内部信息，而是通过该类所提供的方法来实现对内部信息的操作和访问 |
| 实现的目的 | 1.隐藏类的实现细节 2.让使用者只能通过思想预定的方法来访问数据，从而可以在该方法里加入控制逻辑显示对属性不和你访问 3.可以进行数据检查，从而有利于保证对象信息的完整性 4.便于修改，提高代码的可维护性。 |
| 访问控制符 | 1.Private访问控制级别：如果类里的一个成员（包括属性和方法）使用private访问控制符来修饰，则这个成员只能在该类的内部被访问。很显然，这个访问控制符用语修饰属性最合适，使用它来修饰属性就可以吧属性隐藏在内的内部 2.defauit访问控制权限（包访问权限）：如果类里的一个成员（包括属性和方法）或者一个顶级类不适用任何访问控制符修饰，我们就称它是默认访问控制，defauit访问控制的成员或顶级类可以被相同包下其他类访问 3.protected访问控制权限（子类访问权限）：如果一个成员（包括属性和方法）使用protected访问控制符修饰，那么这个成员既可以被同一个包中其他类访问，也可以被不同包中的子类访问。通常情况下，如果使用protrcted来修饰一个方法，通常是希望⑦子类来重写这个方法。 4.public访问权限（公共访问权限）：这是一个最宽松的访问控制级别，如果一个成员（包括属性和方法）或者一个顶级类使用public修饰，这个成员或顶级类就可以被所有类访问，不管访问类和被访问内是否处于同一包中，是否具有父子继承关系 |
| 图表反映级别 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | Private | defauit | Protected | Public | | 同一个类中 | √ | √ | √ | √ | | 同一个包中 |  | √ | √ | √ | | 子类中 |  |  | √ | √ | | 全局范围内 |  |  |  | √ | |
| 程序代码 | Public class Person  {  //将属性使用private修饰，将这些属性隐藏起来  private String name;  private int age;  //提供方法来操作name属性  public void setName(Strin name)  {  //执行合理性校验，要求用户名必须在2~6位之间  If(name.length() > 6 || name.length() < 2)  {  System.out.println("您设置的人名不符合要求");  Return;  }  else  {  This.name = name;  }  }  public String getName()  {  return  this.name;  }  //提供方法来操作age属性  public void setAge(int age)  {  //执行合理性校验，要求用户年龄必须在0~100位之间  If(age >100 || age < 0 )  {  System.out.println("您设置的年龄不合法");  return;  }  else  {  this.age = age;  }  }  public int getAge()  {  return this.age;  }  }  //主函数 public class TestPerson  {  public static void main(String() args)  {  Person p = new Person();  //因为age属性已被隐藏，所以下面语句将出现编译错误  //p.age = 100；  //下面语句编译不会出现错误，但运行时将提示出入的age属性不合法  //程序不会修改p的age属性  p.setAge(1000);  //访问p的age属性也必须通过其对应的getter方法  //因为上面从未成功设置p的age属性，故此处输出0  System.out.println("未能设置age属性时：  "+p.getAge());  //成功修改p 的age属性  p.setAge(30);  //因为上面成功设置了p的age属性，故此处输出30  System.out.println("成功设置了age属性后： "+p.getAge());  //不能直接操作p的name属性，只能通过其对应的setter方法  //因为“李刚”字符串长度满足2~6，所以可以成功设置  p.setName("李刚");  System.out.println(": "+p.getName());    }  } |

继承:

|  |  |
| --- | --- |
| 定义 | Java的继承具有单继承的特点，每个子类只有一个直接父类 |
| 特点 | java的继承需要通富哦extends关键字来思想，思想继承的类被称为子类，被基层的类被称为父类，有的也成为其基类、超类。父类和子类的关系，是一种一般和特殊的关系。 因为子类是一种特殊的父类，影驰父类包含的范围总比子类包含的范围要大，所以可以认为父类是大类，而子类是小类。 |
| 语法格式 | 修饰符 class SubClass extends SupeClass  {  类定义部分  } |
| 程序代码 | public class fruit  {  public double weight;  public void info();  {  System.out.println("我是一个水果！重"+weight+"g!");  }  }  //主程序  public class Apple extends Fruit  {  public static void main(String() args)  {  //创建Apple的对象  Apple a = new Apple();  //Apple对象本身没有weight属性  //因为Apple的父类有weight属性，也可以访问Apple对象的属性  a.weight = 56;  //调用Apple对象的info方法  a.info();  }  }  //输出结果  我是一个水果！重56.0g! |
| supe引用代码 | class BaseClass\  {  public int a = 5;  }  class SubClass extends BaseClass  {  public int a = 7;  public void accessOwner()  {  System.out.println(a);  }  public void accessBase()  {  //通过super类访问方法调用者对应的父类对象  System.out.println(super.a);  }  //主函数  public static void main(String[] args)  {  SubClass sc = new SubClass();  //输出7  sc.accessOwner();  //输出5  scaccessBase();  }  } |

多态

|  |  |
| --- | --- |
| 定义 | 应用变量有两个类型，一个是编译时的类型，一个还是运行时的类型，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，如果编译时类型和运行时类型不一致，就会出现所谓的多态 |
| 程序代码 | class BaseClass  {  public int book = 6;  public void base()  {  System.out.println("父类的普通方法");  }  public void test()  {  System.out.println("父类的被覆盖的方法");  }  }  //继承BaseClass类已经定义主函数  public classSubClass extends BaseClass  {  //重新定义一个book实例属性覆盖父类的bool实例属性  public Sreing book = "轻量级J2EE企业应用实战";  public void test();  {  System.out.println("子类的覆盖父类的方法");  }  public void sub()  {  System.out.println("子类的普通方法");  }  public static void main(String[] args)  {  //下面编译时类型和运行时类型完全一样，因此不存在多态  BaseClass bc = new BaseClass();  //输出6  Systeam.out.println(bc.book);  //下面两次调用将执行BaseClass方法  bc.base();  bc.test();    //下面编译时类型和运行时类型完全一样，因此不存在多态  SubClass sc = new SubClass();  //输出“输出轻量级J2EE企业应用实战”  sc.book();  //下面调用将执行从父类集成到的base方法  sc.base();  //下面调用将执行从当前类的test方法  sc.test();/  /下面编译时类型和运行时类型不一样，多态的发生  BaseClass ploymophicBc =new BaseClass();  //输出6 ---表明访问的是父类属性  System.out.println(ploymophicBc.book);  //下面调用将执行从父类继承到base方法  ploymophicBc.base();  //下面调用将执行从当前类的test方法  ploymophicBc.test();  //因为ploymophicBc的编译类型是BaseClass，BaseClass类没有提供sub方法  //所以下面代码编译会出现错误  //ploymophicBc.sub();  }  } |

instanceof运算符

|  |  |
| --- | --- |
| 定义 | 前一个操作数的通常是一个应用类型的变量，后一个操作数通常是一个类（也可以是接口，可以吧接口理解成一种特殊的类）它用于判断前面的对象是否是后面的类、或者其子类、实现类的实例。如果是，则返回tru否则返回false |
| 程序代码 | public class TestInstanceof  {  public static void main(String[] args)  {  //声明helli时使用object类，则hello的编译类型是object是所有类的父类  //但hello变量四级类型是String  object hello = "Hello";  //String是object类的子类，所以可以进行instanceof运算，返回true  System.out.println("字符串是否是Object类的实例："+(hello instanceof object));  //返回true  System.out.println("字符串死狗是String类的实例："+(hello instanceof String));  //Math是object类的子类，所以可以进行instanceof运算，返回false。  System.out.println("字符串是否是String类的实例："+(hello instanceof Math));  //String实现了comparable接口，所以返回true  System.out.println("字符串是否是Comparable接口实例："+(hello instanceof Comparable));  String a= "Hello";  //String类既不是Math类，也不瘦Math类的父类，所以下面代码编译无法通过  System.out.println("字符串是否是Math类的实例："+(a instanceof Math));  }  } |