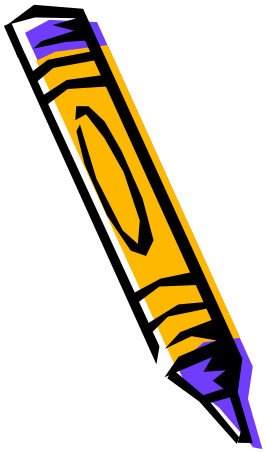
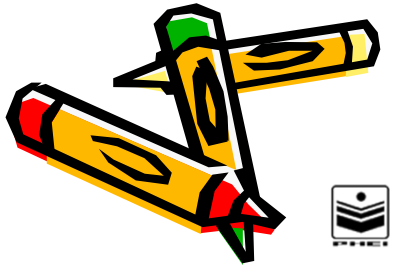
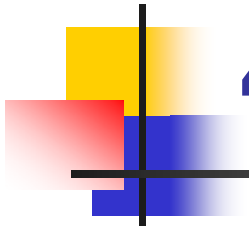


第4章 接口、内部类和Java API基础





4.1 接口

接口：比抽象类还要抽象得彻底的“抽象类”（**JDK7**及以前版本中接口内部只能包含常量和**public**抽象方法，但是**JDK8**以后新增了静态方法和默认方法（**default**修饰的方法）），最抽象。可以更加规范的对实现类进行约束。接口就意味着规范！本质是契约！大家都要遵守！

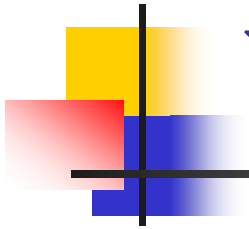
- 一般用于抽取出不容易用继承关系来联系起来的一批类型的相同功能部分（可以抽象出任何类的共同点）。
- 接口就是规范，定义的是一组规则（任何类必须遵守），体现了现实世界“如果你是...你必须能...”的思想。接口存在的意义在于设计和实现的分离，接口分离得最彻底。现代大型软件项目具体需求多变，我们必须先定义好各类规范再开始开发。实际项目中，都是面向接口编程。
- 接口中常量定义时，可以不写**public static final**，缺省有。接口中方法定义时，可以不写**public abstract**，缺省有。接口只定义不变的内容（常量和抽象方法）。
- 一个类可以实现多个接口（**implements**），接口不能**new**一个对象，但是可用于声明引用变量类型。接口可以继承另一个接口（**extends**），而且支持多继承。一个类实现一个接口，必须实现之中所有方法，并且方法只能是**public**的。

4.1 接口与实现接口的类

1. 声明接口

```
[public] interface 接口 [extends 父接口列表 ]
{
   [public] [static] [final] 数据类型 成员变量=常量值;
   [public] [abstract] 返回值类型 成员方法[(参数列表)];
}
public interface Area          //可计算面积接口
{
    public abstract double area();  //计算面积
}
```

接口不能被实例化



可计算周长接口

```
public interface Perimeter
{
   public abstract double perimeter();
                  //抽象方法，计算周长
}
```

2. 声明实现接口的类

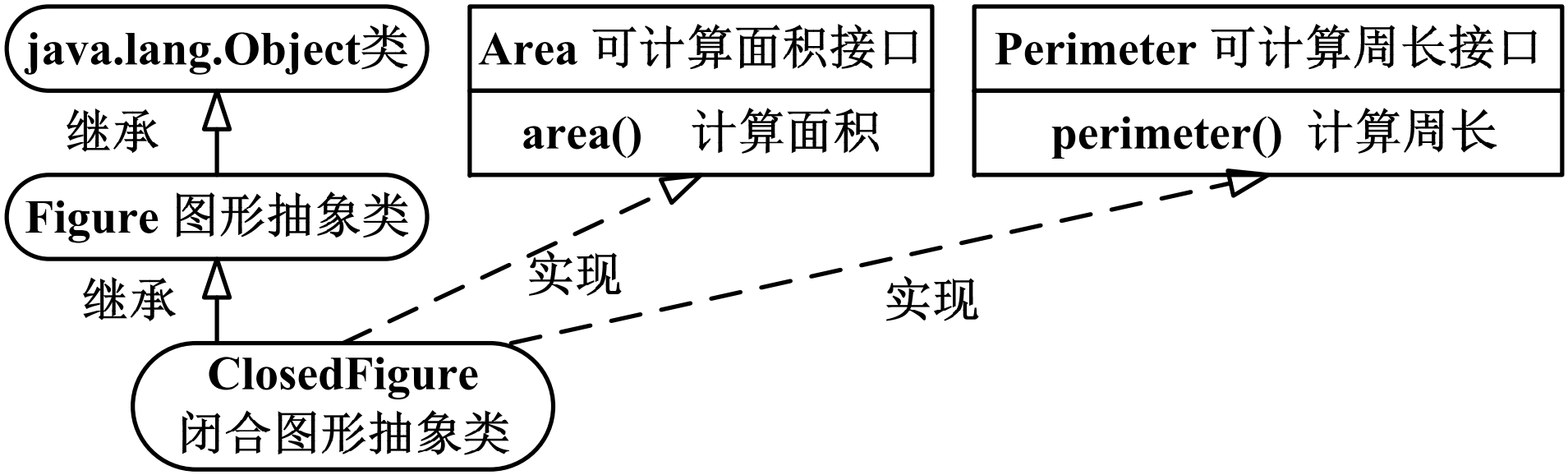
[修饰符] class 类<泛型> [extends 父类]
[implements 接口列表]

例如，

```
public abstract class ClosedFigure
    extends Figure
    implements Area, Perimeter
```

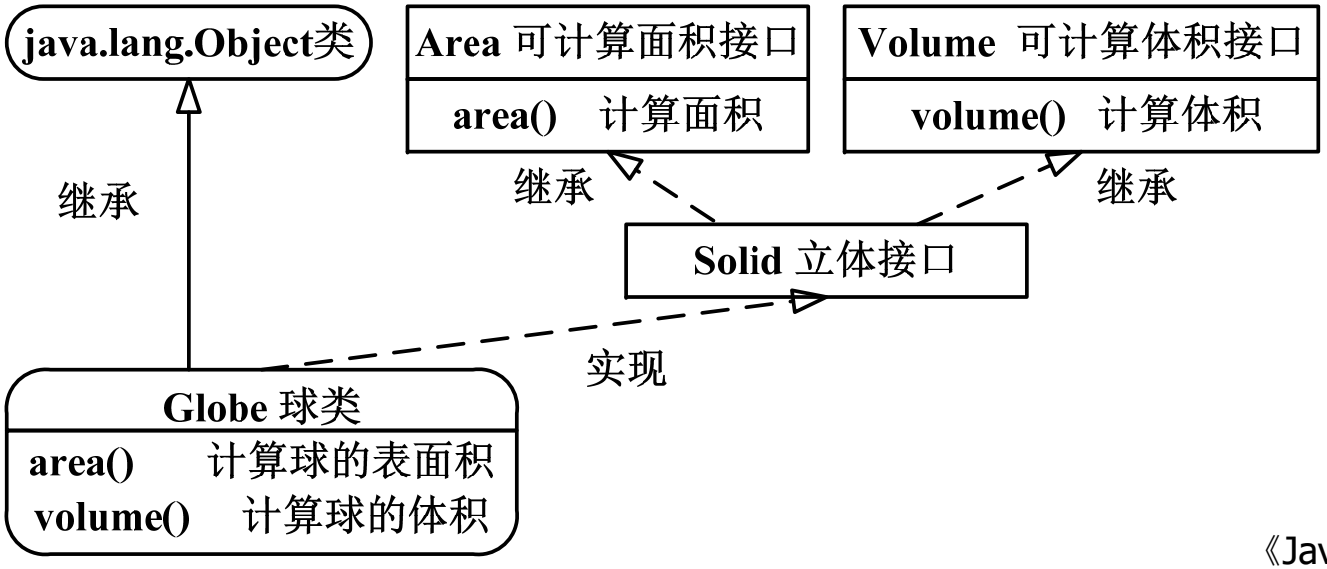
实现接口的非抽象类必须实现所有接口中的所有抽象方法，否则声明为抽象类

图4.1 ClosedFigure类的继承关系



3.接口是多继承的

```
public interface Solid extends Area,Volume
                    //立体接口，继承Area和Volume接口
public class Globe extends Object
        implements Solid  //球类，实现Solid接口
```



4. 接口是引用数据类型

```
ClosedFigure fig = new Ellipse(point,10,20);
                          //父类对象fig引用椭圆子类实例
Area ar = fig;     //Area接口对象ar引用实现Area接口的
                     ClosedFigure类的Ellipse子类实例
ar.area()          //运行时多态

Cylinder cylinder = new Cylinder(fig,10);  //椭圆柱
ar = cylinder;    //ar引用实现Area接口的Cylinder类的实例
Volume vol = cylinder;        //Volume接口对象vol引用实
                          现Volume接口的Cylinder类的实例
ar.area()          //运行时多态
 vol.volume()
```

2. 接口与抽象类的区别

- 抽象类为子类约定方法声明，抽象类可以给出部分实现，包括构造方法等；抽象方法在多个子类中表现出多态性。类的单继承，使得一个类只能继承一个父类的约定和实现。
- 接口为多个互不相关的类约定某一特性的方法声明，在类型层次中表达对象拥有的属性。接口没有实现部分。接口是多继承的。一个类实现多个接口，就具有多种特性，也是多继承的。
- 在项目架构设计时，如何使用抽象类和接口来解决问题，是一个复杂的问题。抽象类更侧重于归纳同一父类的子类的共同特征，如果属性，方法；接口更侧重于定义任意的类有没有相同语义的方法，它是一个一经定义不轻易更改的规范，它的修改在项目中，往往是动一发而牵全身，即使有考虑不周到的地方，也会使用新增接口的形式去弥补。其实**abstract class**表示的是**"is-a"**关系，**interface**表示的是**"like-a"**关系。最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现。

2. 接口与抽象类的区别

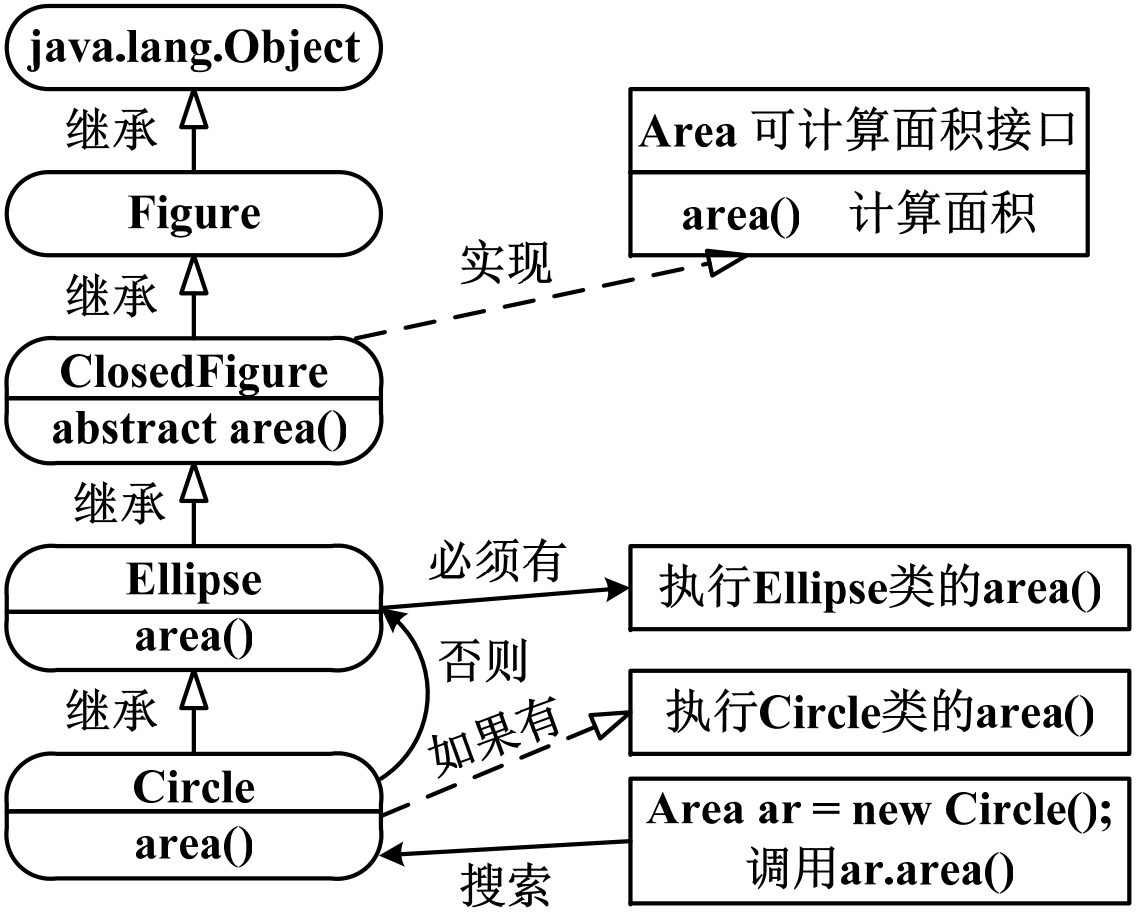
- 接口不能有构造方法，抽象类可以有。抽象类有构造方法，但不能直接用来**new**，只能被子类去调用（包裹实例化）。构造方法是用来在对象初始化前对对象进行一些预处理的，提供了实例化一个具体东西的入口。接口只是声明而已，不一定要进行什么初始化，就算要进行初始化，也可以到实现接口的那一些类里面去初始化。
- 在接口中凡是变量必须是**public static final**，而在抽象类中没有要求。
- 抽象类本质上还是一个类，子类是用关键字 **extends** 来继承它，并扩展的，有非常强的**is-a**的关系。而接口，是被其他类用关键字 **implements** 来实现接口定义的方法的。接口只是定义功能和行为规范，如果一个类实现了一个接口，那么这个类必须遵守这个接口的方法约定，但没有**is-a**的关系。

2. 接口与抽象类的区别

- 接口是一种规范，被调用时，主要关注的是里边的方法，而方法是不需要初始化的，类可以实现多个接口，若多个接口都有自己的构造器，则不好决定构造器的调用次序，构造器是属于类自己的，不能继承，因为是纯虚的，接口不需要构造方法。而抽象类（其最顶端的类也是**Object**，抽象类的父类可以是非抽象类）是具体类的祖先，即使追溯到**Object**，也总要干些初始化的工作，抽象类虽然是不能直接实例化，但实例化子类的时候，就会初始化父类，不管父类是不是抽象类都会调用父类的构造方法，初始化一个类，先初始化父类，没有说要初始化接口。
- 类只能单继承，但可以实现多个接口；接口则可以多继承。

3. 单继承和多继承

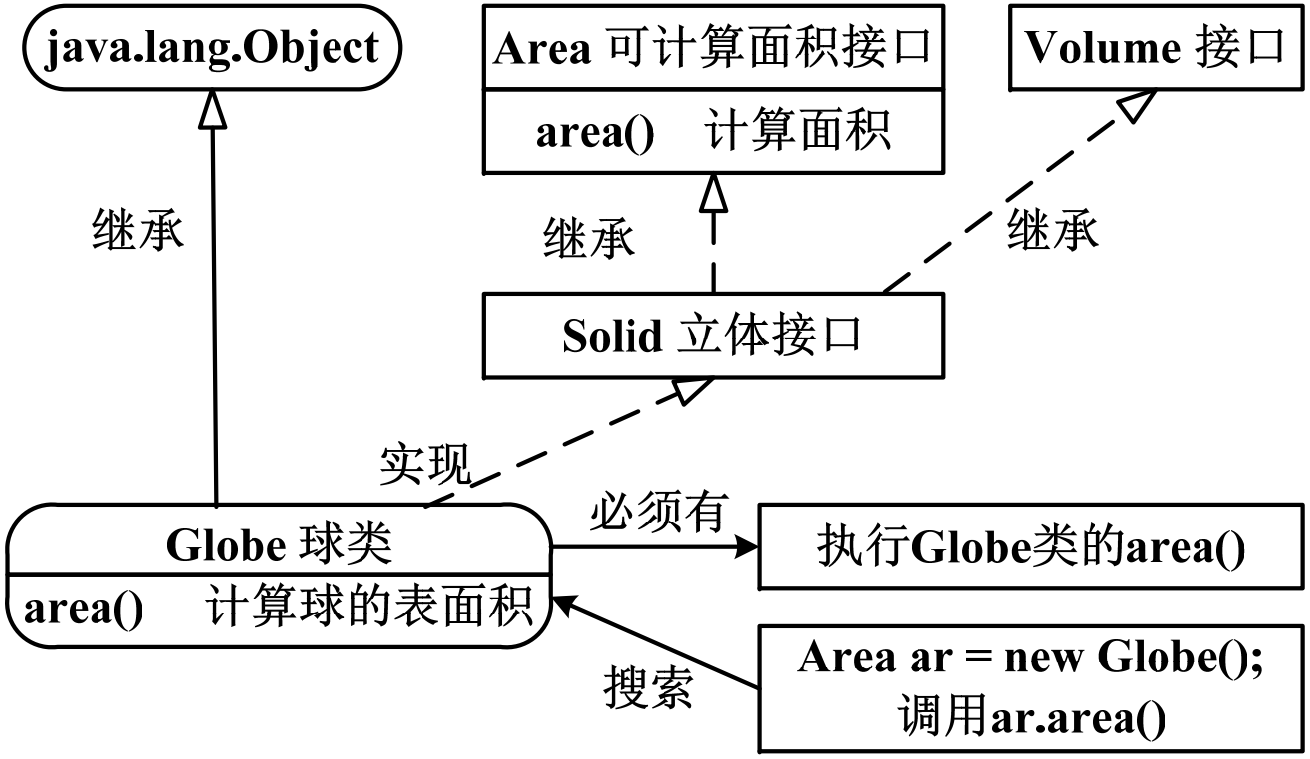
1. 类的单继承的优点



（a）类的单继承使Java搜索匹配执行方法的路径是线性的

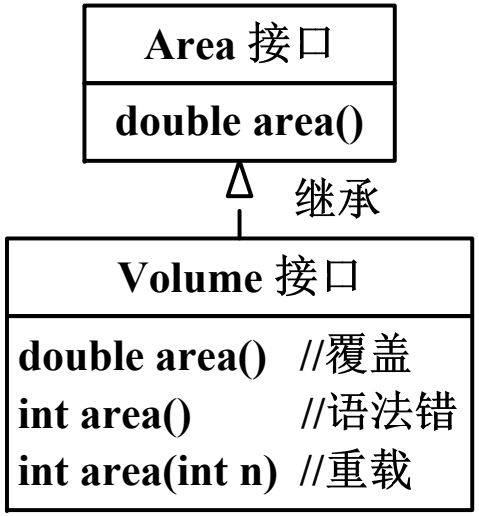
3. 单继承和多继承

1. 类的单继承的优点

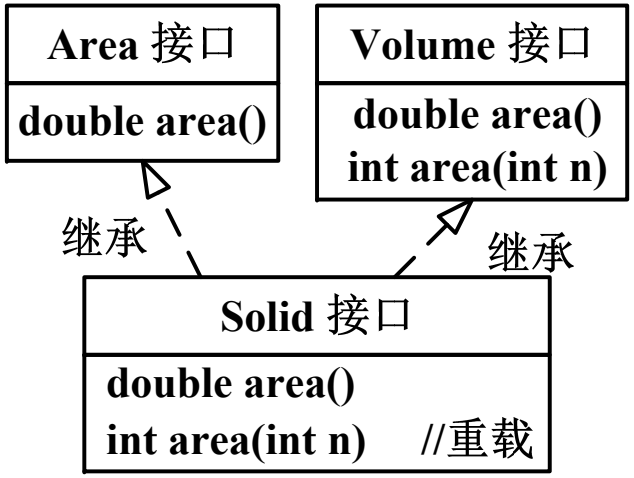


（b）接口的多继承对方法的运行时多态搜索没有影响

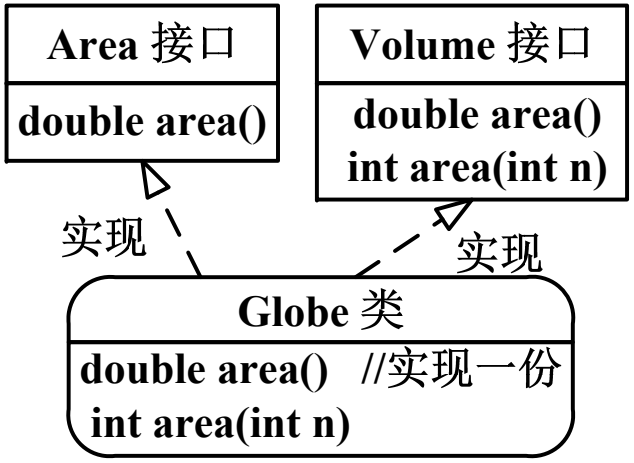
(2)接口的多态性



（a）接口继承



（b）接口多继承



（c）类实现多接口

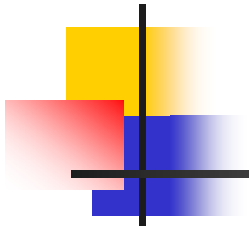
4.2 内部类和内部接口

```
public class Pixel    //像素类，外层类型，外部类
{   public static interface ColorConstant
                        //颜色常量接口，静态内部接口，类型嵌套
    public static class Color extends Object
          implements ColorConstant
                              //颜色类，静态内部类
}
```

类型嵌套：静态内嵌类型

```
Pixel.Color color = new Pixel.Color(255,255,255);
```

对象嵌套：实例内嵌类型

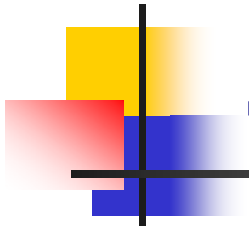


1. 作为类型的特性

1. 内嵌类型不能与外层类型同名。
2. 内部类中可以声明成员变量和成员方法。
3. 内部类可以继承父类或实现接口。
4. 可以声明内部类为抽象类，该抽象类必须被其他内部类继承；内部接口必须被其他内部类实现。

2. 作为成员的特性

1. 使用点运算符“.”引用内嵌类型：
   外层类型.内嵌类型
   Pixel.Color
2. 内嵌类型具有类中成员的**4**种访问控制权限。当内部类可被访问时，才能考虑内部类中成员的访问控制权限。
3. 内嵌类型与其外层类型彼此信任，能访问对方的所有成员。
4. 内部接口总是静态的。内部类可声明是静态的或实例的，静态内部类能够声明静态成员，但不能引用外部类的实例成员；实例内部类不能声明静态成员。



2. 作为成员的特性

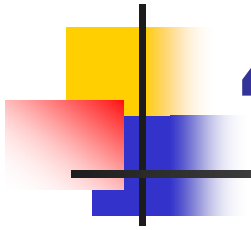
5. 在实例内部类中，使用以下格式引用或调用外部类当前实例的成员变量或实例成员方法：

外部类.this.成员变量

//引用外部类当前实例的成员变量

外部类.this.实例成员方法(参数列表)

//调用外部类当前实例的成员方法



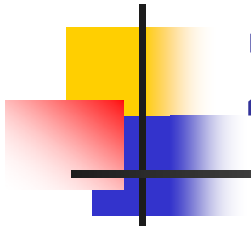
4.3 Java API基础

1. 4.3.1 java.lang包中的基础类库
2. 4.3.2 java.util包中的工具类库

4.3.1 java.lang包中的基础类库

1. **Object类**

```
package java.lang;
public class Object
{
   public Object()                          //构造方法
   public final Class<?> getClass();     //返回当前对象所在的类
   public boolean equals(Object obj)
                                          //比较当前对象与obj是否相等
   public String toString()          //返回当前对象的信息字符串
   protected void finalize() throws Throwable       //析构方法
}
```



2. Math数学类

```
public final class Math extends Object
{
   public static final double E = 2.71828182845904523536; //常量
   public static final double PI = 3.14159265358979323846;//π

   public static double abs(double a)                    //求绝对值
   public static double random()    //返回一个0.0~1.0之间的随机数
    public static double pow(double a, double b)    //返回a的b次幂
    public static double sqrt(double a)              //返回a的平方根值
    public static double sin(double a)               //返回a的正弦值
}
```

3. Comparable可比较接口

```
public interface Comparable<T>
{
    int compareTo(T cobj)  //比较对象大小
}
```

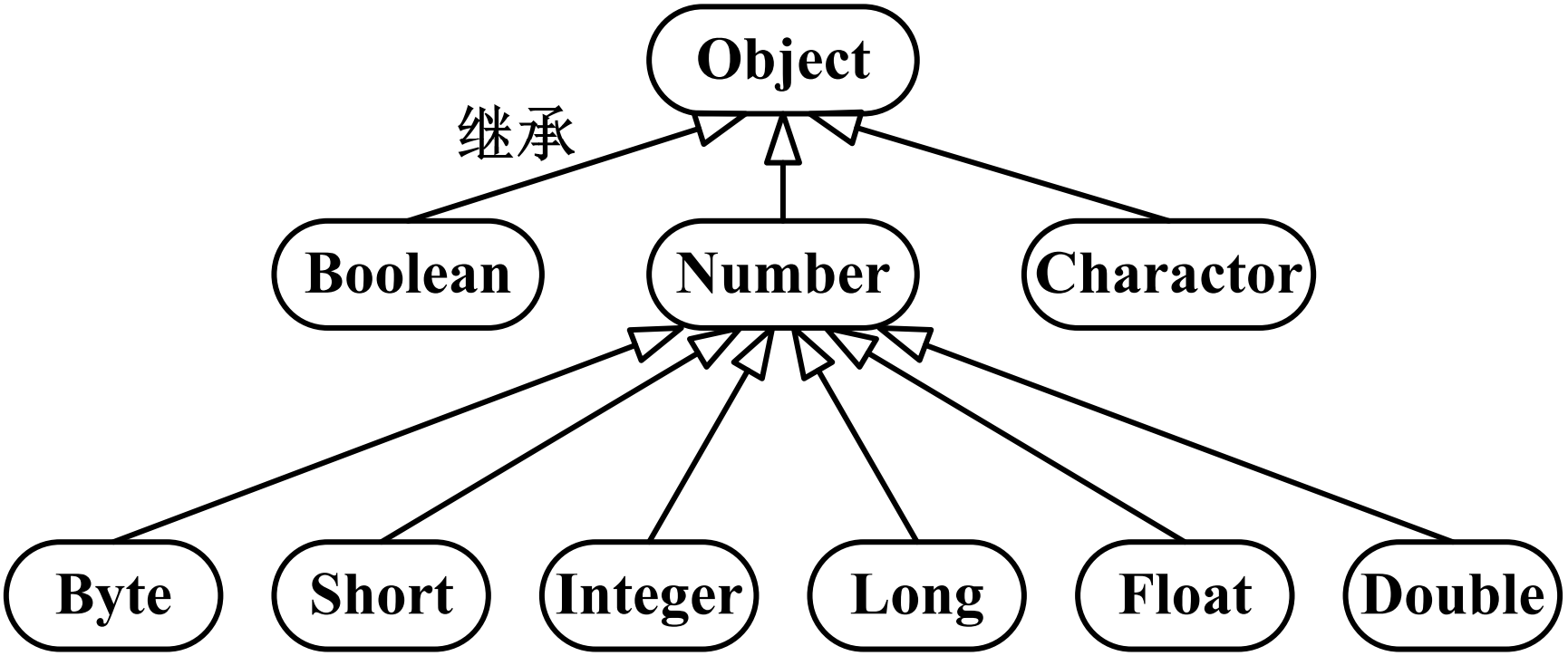
其中，<T>是Comparable接口的参数，表示一个类。

MyDate类对象比较大小

```
public class MyDate implements
        Comparable<MyDate>
{
   public int compareTo(MyDate d)
                //约定比较日期大小的规则，返回-1、0、1
   {
      if (this.year==d.year && this.month==d.month
     && this.day==d.day)
          return 0;
      return (this.year>d.year || this.year==d.year &&
     this.month>d.month || this.year==d.year &&
     this.month==d.month && this.day>d.day) ? 1 : -1;
   }
}
```

4. 基本数据类型的包装类

8个Byte、Short、Integer、Long、Float、Double、Character、Boolean。

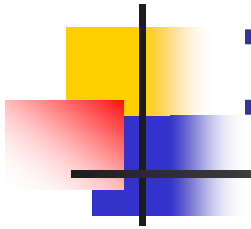


基本数据类型的包装类（Wrapper Class）

- 为什么需要包装类？基于运行效率考虑，**JAVA**中的基本数据类型不是面向对象的。而在实际应用中可能有时又需要将基本数据转化为对象以便于统一操作。（比如：集合操作中，我们就需要将基本数据类型转化为对象！）
- 包装类均位于**java.lang**，对应关系：**Byte-byte; Boolean-boolean; Short-short; Character-char; Integer-int; Long-long; Float-float; Double-double**。
- 包装类的作用：提供字符串、基本数据类型、包装类对象之间互相转化的方式；包含每种基本数据类的相关属性如最大值、最小值等等。

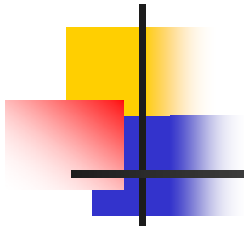
自动装箱和自动拆箱（auto-boxing/unboxing）

- 自动装箱：基本数据类型就自动的封装到与它相同类型的包装中。如：**Integer i = 100;** 本质上就是编译器编译时自动为我们添加了 **Integer i = new Integer(100);**
- 自动拆箱：包装类对象自动转化为基本数据类型。如：**int a = new Integer(100);** 本质上就是编译器编译时自动为我们添加了**int a = new Integer(100).intValue();**
- 缓存问题：**[-128,127]**之间的数对应的包装类对象，仍然当做基本数据类型来处理；一旦遇到一个这个之间的数（默认为这些小的数使用频率会很高），把他包装成一个对象后，就缓存起来，下次如果又要包装一个这个数的对象，则去看是否已经有这个对象，有就直接拿来使用，这样可以节省内存空间、提高效率（享元模式）。
- 享元模式：有很多小对象，它们的大部分属性相同，这时可以把它们变成一个对象，那些相同的属性为对象的内部状态，那些不同的属性可以变为方法的参数，由外部传入。例：**-128~127**内的相同整数自动装箱为同一个对象。



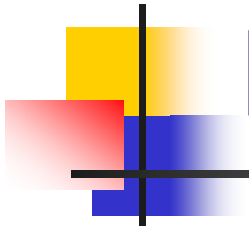
Integer

```
public final class Integer extends Number
        implements Comparable<Integer>
{
   public static final int MIN_VALUE=0x80000000;//最小值-2^31
   public static final int MAX_VALUE = 0x7fffffff;  //最大值2^31-1
   private final int value;                    //私有最终变量，构造时赋值

   public Integer(int value)                        //构造方法
   public Integer(String s) throws NumberFormatException
   public static int parseInt(String s) throws
     NumberFormatException      //将字符串转换为整数，静态方法
```



Integer

```
public String toString()    //覆盖Object类中方法
public static String toBinaryString(int i)
            //将i转换成二进制字符串，i≥时，省略高位0
public static String toOctalString(int i)
            //将i转换成八进制字符串，i≥时，省略高位0
public static String toHexString(int i)
           //将i转换成十六进制字符串
public boolean equals(Object obj)
                          //覆盖Object类中方法
 public int compareTo(Integer iobj)
                  //比较两个对象值大小，返回–1、0或1
}
```



Double类

```
public final class Double extends Number
        implements Comparable<Double>
{
    public Double(double value)
    public Double(String s) throws
      NumberFormatException
    public static double parseDouble(String s) throws
      NumberFormatException //将串s转换为浮点数
    public double doubleValue()
                                  //返回当前对象中的浮点数值
}
```

5. String字符串类

```
public final class String extends Object
  implements java.io.Serializable, Comparable<String>,
        CharSequence
{
    private final char value[];                    //字符数组，最终变量
   public String()                                 //构造方法
   public String(String original)
   public String toString()                        //覆盖Object类中方法
   public int length()                             //返回字符串的长度
   public boolean equals(Object obj)               //比较字符串是否相等
   public boolean equalsIgnoreCase (String s)//忽略字母大小写
   public int compareTo(String s)                  //比较字符串的大小
   public int compareToIgnoreCase(String str)
}
```

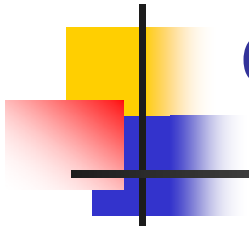
7. Class类

```
public final class Class<T>
{
   public String getName() //返回当前类名字符串
    public Class<? super T> getSuperclass();
                                    //返回当前类的父类
    public Package getPackage()
                                    //返回当前类所在的包
}
```

```
this.getClass().getName()
this.getClass().getSuperclass().getName()
this.getClass().getPackage().getName()
```

8. System系统类

```
public final class System extends Object
{
   public final static InputStream in = nullInputStream();
   public final static PrintStream out = nullPrintStream();
   public final static PrintStream err = nullPrintStream();
   public static native viod arraycopy(Object src, int src_pos,
      Object dst, int dst_pos, int length)        //复制数组
   public static void exit(int status)            //结束当前运行的程序
   public static native long currentTimeMillis();
      //获得当前日期和时间，返回从1970-1-1 00:00:00开始至当前时
      间的累计毫秒数
   public static Properties getProperties()  //获得系统全部属性
   public static String getProperty(String key)  //获得指定系统属性
}
```



9. Runtime运行时类

```
public class Runtime extends Object
{
    public static Runtime getRuntime()
        //返回与当前应用程序相联系的运行时环境
    public long totalMemory()
                    //返回系统内存空间总量
    public long freeMemory()
                    //返回系统内存剩余空间的大小
}
```

4.3.2 java.util包中的工具类库

1. 日期类

① Date日期类

```
public class Date extends Object  implements
      java.io.Serializable, Cloneable, Comparable<Date>
{
   public Date()                  //获得系统当前日期和时间的Date对象
   {
      this(System.currentTimeMillis());
   }
   public Date(long date)              //以长整型值创建Date对象
   public int compareTo(Date date)
                                       //比较日期大小，返回0、1、-1
}
```

(2) Calendar类

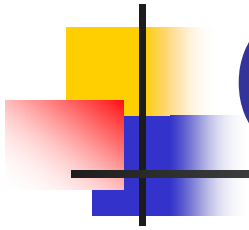
```
public abstract class Calendar extends Object
        implements Serializable, Cloneable, Comparable<Calendar>
{
   public static final int YEAR                 //年，常量
   public static final int MONTH                //月
   public static final int DATE                 //日
   public static final int HOUR                 //时
   public static final int MINUTE               //分
   public static final int SECOND               //秒
   public static final int MILLISECOND          //百分秒
   public static final int DAY_OF_WEEK          //星期
}
```

(2) Calendar类

```
public abstract class Calendar extends Object
        implements Serializable, Cloneable, Comparable<Calendar>
{
   public static Calendar getInstance()        //创建实例
   public int get(int field)                   //返回日期
   public final Date getTime()                 //返回对象中的日期和时间
   public final void setTime(Date date)    //设置对象的日期和时间
   public final void set(int year, int month, int date)
   public final void set(int year, int month, int date, int hour, int
      minute)
}
```

例如，Calendar now = Calendar.getInstance();   //获得实例

int year =now.get(Calendar.YEAR);   //获得对象中的年份值



(3) GregorianCalendar类

```
public class GregorianCalendar extends Calendar
{
   public GregorianCalendar()          //以当前日期时间创建对象
   public GregorianCalendar(int year, int month, int day)
   public GregorianCalendar(int year, int month, int day, int
      hour, int minute, int second)
   public boolean isLeapYear(int year)      //判断是否闰年
}
```

//例3.2 MyDate类中，获得当前日期对象的年份值

```
thisYear=new GregorianCalendar().get(Calendar.YEAR);
```

【例4.3】 月历。

2.Comparator比较器接口

```
public interface Comparator<T>
{
    public abstract boolean equals(Object obj);
                                //比较两个比较器对象是否相等
    public abstract int compare(T cobj1, T
     cobj2);        //指定比较两个对象大小的规则
}
```

【例4.4】Person类的多种比较规则。

(1) 约定多种比较对象大小的规则

(2) 约定多种比较对象相等的规则

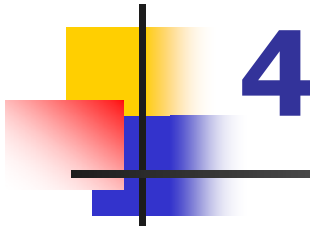
3. Arrays数组类

1. 排序

```
public static void sort(Object[] a)
public static <T> void sort(T[] a,
                    Comparator<? super T> c)
```

2. 二分法（折半）查找

```
public static int binarySearch(Object[] a,
    Object key)
public static <T> int binarySearch(T[] a,
   T key, Comparator<? super T> c)
```



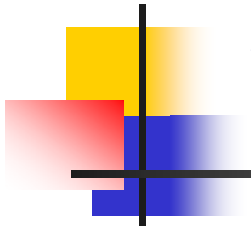
4.4 泛型

1. 泛型声明

[修饰符] **class** 类**<类型参数列表>**
[extends 父类**]** **[implements** 接口列表**]**

[public] interface 接口**<类型参数列表>**
[extends 父接口列表**]**

[public] [static] **<类型参数列表>** 返回值类型 方法**([**参数列表**])** **[throws** 异常类列表**]**

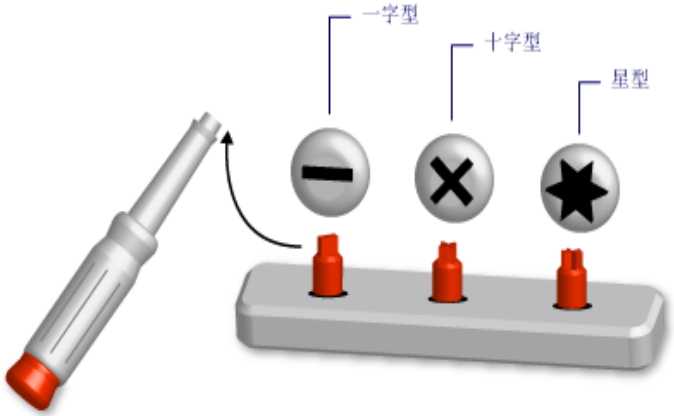


泛型设计的起因

- 起因:JDK1.4以前类型不明确
  - 装入集合的类型都被当作Object对待，从而失去自己的实际类型。
  - 从集合中取出时往往需要转型，效率低，容易产生错误。
  - 方案:在定义集合的时候同时定义集合中对象的类型
  - 作用:
  - **模板**:提高代码的重用率
  - **安全**:在编译的时候检查类型安全
  - **省心**:所有的强制转换都是自动和隐式的

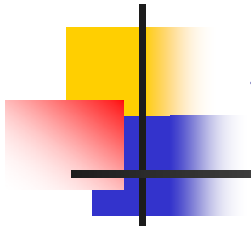
泛型概念

- 概念:泛型就是参数化类型。
  - **适用于对多种数据类型执行相同功能的代码。**
  - 泛型中的类型在使用时指定。
  - 泛型归根到底就是“**模板**”。
- 如:可拆卸刀头的螺丝刀
  - 检查需要拧动的螺丝，
  - 根据螺丝选择适合刀头（一字、十字、星形），
  - 将正确的刀头插入到螺丝刀柄上后，



您就可以使用螺丝刀执行完全相同的功能，即拧螺丝。

- 泛型主要适用在集合中



任意化问题

- 引出:学生成绩有三种情况
  - 1、整数
  - 2、小数
  - 3、字符串
- 任意化: Object是所有类的根类，
  但是具体类使用的时候;需要类型强制转换的
  - **多态:Object可以接受任意类型。**
  - **缺点:**
    - **需要类型转换;**
    - **需要类型检查;**
    - **需要处理转换错误异常**
- 泛型:使用泛型时,在实际使用之前类型就已经确定了，不需要强制转换。

传统任意化实现举例

Student 使用Object 代码片段

```
public class Student {
   private Object javase;
   public Student() {
   }
   public Student(Object javase) {
      this.javase =javase;
   }
   public Object getJavase() {
      return javase;
    }
   public void setJavase(Object javase) {
       this.javase = javase;
   }
}
```

```
public static void main(String[] args) {
   //存入整数 int -->Integer -->Object
   Student stu = new Student(80);

    Object javase =stu.getJavase();
   //类型检查  处理转换异常
   if(javase  instanceof Integer){
    //强制类型转换
     Integer javaseScore =(Integer) javase ;
   }
}
```

泛型字母

- **形式类型参数（formal type parameters ）即泛型字母**
- 命名:泛型字母可以随意指定，尽量使用单个的大写字母（有时候多个泛型类型时会加上数字，比如T1，T2
- 常见字母 （见名知意）
  - T Type
  - K V Key Value
  - E Element
  - 当类被使用时，会使用具体的实际类型参数（**actual type argument**）代替

HashMap源码 使用 K V

```
public class HashMap<K,V> extends
AbstractMap<K,V> implements Map<K,V>…{
}
```

ArrayList源码 使用 E

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>…{


}
```

自定义泛型类

- 定义类时使用泛型, < >
- **定义模板**

```
class 类名<字母>{
        private 字母 属性名;
        …setter与getter.
}
```

a) 不能使用在基本类型上

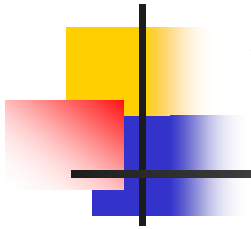
b) 不能使用在静态属性上

Student 使用泛型

```
// 此处可以随便写标识字母，T是type的简称
public class Student<T> {
   private T javase;
   public Student(T javase) {
            this.javase = javase;
    }
   public void setJavase(T javase) {
            this.javase=javase;
```

```
public static void main(String[] args) {
   //使用时指定类型(引用类型Integer)
    Student<Integer> stu = new Student<Integer> ();
```

```
   //使用时指定类型(引用类型String)
    Student<String> stu = new Student<String> ();
   //1、安全:类型检查
    stu.setJavase("优秀");
   //2、省心:类型转换
    String javase =stu.getJavase();
```



泛型接口

定义接口时使用泛型

泛型接口实例

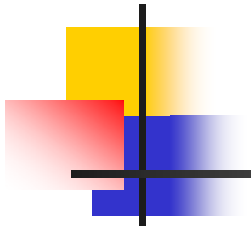
public interface Comparator<T> {
　　void compare(T t);
}

注意:接口中泛型字母只能使用在方法中，不能使用在全局常量中

自定义泛型方法

- 定义方法时: <字母>
- 注意:
  - 泛型方法可以在非泛型类中

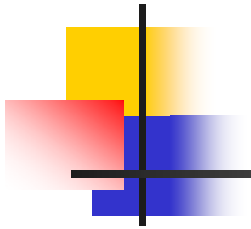
```
public class TestMethod {
        public static void main(String[] args) {
                test("a"); //T -->String
        }
        //定义泛型方法
        public static <T> void test(T a){
                System.out.println(a);
        }
        // extends <=
        public static <T extends Closeable> void test(T... a){
                for(T temp:a){
                        try {
                          if(null!=temp){  temp.close();}
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```



了解泛型

- ArrayList<E>类定义和ArrayList<Integer>类引用中涉及如下术语：
  - 整个称为ArrayList<E>泛型类型
  - ArrayList<E>中的E称为类型变量或类型参数
  - 整个ArrayList<Integer>称为参数化的类型
  - ArrayList<Integer>中的Integer称为类型参数的实例或实际类型参数
  - ArrayList<Integer>中的<>念着typeof
  - ArrayList称为原始类型
- 参数化类型与原始类型的兼容性：
  - 参数化类型可以引用一个原始类型的对象，编译报告警告，例如，
    Collection<String> c = new Vector();//可不可以，编译器说了算
  - 原始类型可以引用一个参数化类型的对象，编译报告警告，例如，
    Collection c = new Vector<String>();//原来的方法接受一个集合参数，新的类型也要能传进去
- 参数化类型不考虑类型参数的继承关系：
  - Vector<String> v = new Vector<Object>(); //错误!///不写<Object>没错，写了就是明知故犯
  - Vector<Object> v = new Vector<String>(); //也错误!
- 思考题：下面的代码会报错误吗？

```
Vector v1 = new Vector<String>();
Vector<Object> v = v1;
```



了解泛型

- 编译器不允许创建泛型变量的数组。即在创建数组实例时，数组的元素不能使用参数化的类型，例如，下面语句有错误：

  Vector<Integer> vectorList[] = new Vector<Integer>[10];

- 原因是因为数组提供了运行时的类型安全，但是没有编译时的类型安全检测；而泛型提供了编译时类型安全检查，运行时进行类型擦除。因此，数组和泛型不能很好的混合使用。创建泛型数组(new List<E>[]),创建参数化类型数组(new List<String>[]),创建类型参数数组(new E())都是非法的。这些类型称作”不可具体化类型"。唯一可具体化的参数化类型是无限制的通配符类型，如List<?>和Map<?,?>。虽然不常用，但是创建无限制通配符类型数组是合法的。
- 此外，由于每次调用可变参数都会创建一个数组来存放可变的参数。如果这个数组的元素类型是不可具体化的，就会得到一个警告。对于这个警告的处理除了@SuppressWarning把它禁止之外，就是避免在API中混合使用泛型与可变参数。
- 最后，当获得一个创建泛型数组错误时，最好的解决办法就是优先使用集合类型List<E>或者Object[],而不是数组类型E[]。
- 不过，可以以下面的方式使用参数化类型数组。

  List<Integer>[] ls = new ArrayList[5];

体验泛型（以集合举例）

- Jdk 1.5以前的集合类中存在什么问题

```
ArrayList collection = new ArrayList();
collection.add(1);
collection.add(1L);
collection.add("abc");
int i = (Integer) collection.get(1);//编译要强制类型转换且运行时出错！
```

- Jdk 1.5的集合类希望你在定义集合时，明确表示你要向集合中装哪种类型的数据，无法加入指定类型以外的数据

```
ArrayList<Integer> collection2 = new ArrayList<Integer>();
collection2.add(1);
/*collection2.add(1L);
collection2.add( “abc” );*///这两行代码编译时就报告了语法错误
int i2 = collection2.get(0);//不需要再进行类型转换
```

- 泛型非常适用于对所有的类型执行相同功能的代码，例如集合类。

- 泛型是提供给javac编译器使用的，可以限定集合中的输入类型，让编译器挡住源程序中的非法输入，编译器编译带类型说明的集合时会去除掉“类型”信息，使程序运行效率不受影响，对于参数化的泛型类型，getClass()方法的返回值和原始类型完全一样。由于编译生成的字节码会去掉泛型的类型信息，只要能跳过编译器，就可以往某个泛型集合中加入其它类型的数据，例如，用反射得到集合，再调用其add方法即可。

Java中的泛型思想类源于C++中的模板函数

- 如下函数的结构很相似，仅类型不同：
  - int add(int x,int y) {
    return x+y;
    }
  - float add(float x,float y) {
    return x+y;
    }
  - double add(double x,double y) {
    return x+y;
    }
- C++用模板函数解决，只写一个通用的方法，它可以适应各种类型，示意代码如下：

```
template<class T>
T add(T x,T y) {
        return (T) (x+y);
}
```

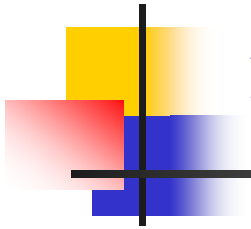
泛型继承、实现

- 父类为泛型类，子类继承时:
  - 父类擦除|指定类型，子类按需编写
  - 父类存在泛型，子类必须>=
  - 属性及方法类型:随位置而定
- 接口为泛型接口
  - 与继承同理,重写方法随接口而定

```
public abstract class Father<T1,T2> {
        T1 age;
        public abstract void test(T2 name);

}
```

```
//父类擦除|指定类型，子类按需编写
class C1<T1,T2> extends Father{
        public void test(Object name) {
        }
}
class C3<E> extends Father<String,Integer>{
        E name;
        public void test(Integer name) {
        }
}
```

```
//父类存在泛型，子类必须>=
class C4<T1,T2> extends Father<T1,T2>{
        public void test(T2 name) {
        }
}
```



泛型擦除

- 定义:泛型擦除是指在继承(实现)或使用时没有指定具体的类型
- 特点:一旦擦除之后按Object处理
  - 依然存在警告，加上Object可以去除，但是有些画蛇添足
  - 不完全等同于Object ,编译不会类型检查

泛型擦除 (代码片段)

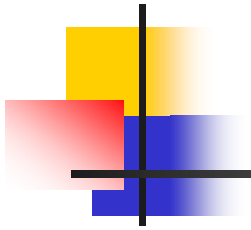
```
public static void main(String[] args) {
        //1、泛型擦除，但是存在警告
        Student stu = new Student();
        stu.setJavaScore("af"); //以Object处理

        //2、消除警告 使用 Object
       Student<Object> student = new Student<Object>();

       //3、不完全等同于Object ,编译不会类型检查
       test(stu); //正确: 擦除，编译通过，不会类型检查
      //test(student) ;//错误 ， Object 编译检查
 }
public static  void test(Student<Integer> a){
 }
```

泛型中的？通配符

- ？**通配符(Wildcards)**
  - T 、 K、 V、 E 等泛型字母为**有类型**，类型参数赋予具体的值
  - ？ **未知类型** 类型参数赋予不确定值，任意类型
  - 只能用在声明类型 、方法参数上，不能用在定义泛型类上

通配符 (代码片段)

```
//此处T不能够换成?
public class Student<T> {
    T score;
    public static void main(String[] args) {
        //通配符 :用在声明类型
        Student<?> stu = new Student<String>();
        test(new Student<Integer>());
    }
    //通配符 :用在声明方法参数  接收信息
    public static void test(Student<?> stu){
        //获取信息
        System.out.print(stu.score);
    }
}
```



泛型中的？通配符扩展

- 限定通配符的上边界：
  - 正确：Vector<? extends Number> x = new Vector<Integer>();
  - 错误：Vector<? extends Number> x = new Vector<String>();
- 限定通配符的下边界：
  - 正确：Vector<? super Integer> x = new Vector<Number>();
  - 错误：Vector<? super Integer> x = new Vector<Byte>();
- 提示：
  - 限定通配符总是包括自己。
  - ?只能用作引用，不能用它去给其他变量赋值

    Vector<? extends Number> y = new Vector<Integer>();

    Vector<Number> x = y;

    上面的代码错误，原理与Vector<Object > x11 = **new** Vector<String>();相似，只能通过强制类型转换方式来赋值。

泛型中的？通配符

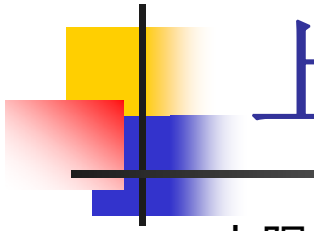
- 问题：
  - 定义一个方法，该方法用于打印出任意参数化类型的集合中的所有数据，该方法如何定义呢？
- 错误方式：

```
public static void printCollection(Collection<Object> cols) {
        for(Object obj:cols) {
                System.out.println(obj);
        }
        /* cols.add("string");//没错
         cols = new HashSet<Date>();//会报告错误！ */
}
```

- 正确方式：

```
public static void printCollection(Collection<?> cols) {
        for(Object obj:cols) {
                System.out.println(obj);
        }
        //cols.add("string");//错误，因为它不知自己未来匹配就一定是String
        cols.size();//没错，此方法与类型参数没有关系
         cols = new HashSet<Date>();
    }
```

- 总结：
  - 使用?通配符可以引用其他各种参数化的类型，?通配符定义的变量主要用作引用，可以调用与参数化无关的方法，不能调用与参数化有关的方法。



上限

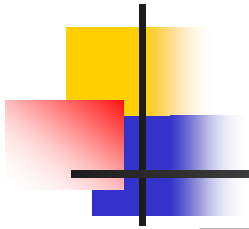
- 上限 extends :指定的类型必须是继承某个类，或者实现某个接口(不是用implements)，即<=如
  - ? extends Fruit
  - T extends List

- 不能添加信息
- 存在以下规则,如
  - List<Fruit > 满足 List<? extends Fruit>
  - List<? extends Apple> 满足 List<? extends Fruit>
  - List<?>等同List<? extends Object>

```
public class Foo<T extends List> {
    public static void main(String[] args) {
        //上限
        Foo<ArrayList> f =new Foo<ArrayList>;
        Foo<LinkedList> f2 =new Foo<LinkedList>;
    }
}
```

```
public void test(List<? extends Fruit> list){
   list.add(new Fruit("f"));
   list.add(new Pear("p"));
   list.add(new Apple("a"));
}
```

以上代码无法通过编译。为什么呢？？



下限

- 下限 super：指定的类型不能小于操作的类， 即>=
  - T super Apple
  - ？ super Apple

- 不能添加父对象
- 存在以下规则,如
  - List<? super Fruit> 满足 List<? super Apple>
  - List<Fruit> 满足 List<? super Fruit>

泛型嵌套

- 稍微复杂一些，从外到内拆分

```
public class Student<T> {
        T score;
}
```

```
public class Whut <T>{
 T stu ;
}
```

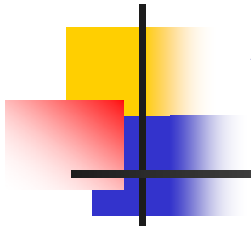
```
public static void main(String[] args) {
    //泛型的嵌套
    Bj<Student<String>> room =new
Whut<Student<String>>();
        //从外到内拆分
        room.stu = new Student<String>();
        Student<String> stu = room.stu;
        String score =stu.score;
        System.out.println(score);
}
```

其他

- 泛型没有多态
- 没有泛型数组（?）
- jdk7简化泛型

```
public class OthersApp {
        public static void main(String[] args) {
          //没有泛型数组 Student<String>[] arr2 = new Student<String>[10];
           //泛型没有多态
           //A<Fruit> f = new A<Apple>();
          //test(new A<Apple>());
        }
        public static void test(A<Fruit> f){   }
        public static A<Fruit>  test2(){
                //return (A<Fruit>)(new A<Apple>());
                return null;
        }}
```

```
public class Jdk7{
    public static void main(String[] args) {
        //1.7之前两边都需要指定类型
        List<String>  arrList= new ArrayList<String>();
        //1.7中使用泛型，声明类型即可，使用/创建时不用指定类型
        List<String>  arrList2= new ArrayList<>();
  }
}
```



课堂作业

- 编写一个泛型方法，自动将Object类型的对象转换成其他类型。
- 采用自定泛型方法的方式打印出任意参数化类型的集合中的所有内容。
- 定义一个方法，把任意参数类型的集合中的数据安全地复制到相应类型的数组中。

类型参数的类型推断

- 编译器判断泛型方法的实际类型参数的过程称为类型推断。
- 根据调用泛型方法时实际传递的参数类型或返回值的类型来推断，具体规则如下：
  - 当某个类型变量只在整个参数列表中的所有参数和返回值中的一处被应用了，那么根据调用方法时该处的实际应用类型来确定，这很容易凭着感觉推断出来，即直接根据调用方法时传递的参数类型或返回值来决定泛型参数的类型，例如：

    swap(new String[3],3,4) → static <E> void swap(E[] a, int i, int j)
  - 当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型都对应同一种类型来确定，这很容易凭着感觉推断出来，例如：

    add(3,5) → static <T> T add(T a, T b)
  - 当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型对应到了不同的类型，且没有使用返回值，这时候取多个参数中的最大交集类型，例如，下面语句实际对应的类型就是Number了：

    fill(new Integer[3],3.5f) → static <T> void fill(T[] a, T v)
  - 当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型对应到了不同的类型，并且使用返回值，这时候优先考虑返回值的类型，例如，下面语句实际对应的类型就是Integer了，编译将报告错误，将变量x的类型改为float，对比eclipse报告的错误提示，接着再将变量x类型改为Number，则没有了错误：

    int x =(3,3.5f) → static <T> T add(T a, T b)
  - 参数类型的类型推断具有传递性，下面第一种情况推断实际参数类型为Object，编译没有问题，而第二种情况则根据参数化的Vector类实例将类型变量直接确定为String类型，编译将出现问题：

    copy(new Integer[5],new String[5]) → static <T> void copy(T[] a,T[] b);

    copy(new Vector<String>(), new Integer[5]) → static <T> void copy(Collection<T> a , T[] b);