

Java程序设计 (android)

2019.7.18

isszym sysu.edu.cn

[SE8 SE10 apkbus runoob
download](#)

目录

[C++编译器和Java虚拟机](#)

[Java的特点](#)

[JRE和JDK](#)

[第一个Java程序](#)

[数据类型](#)

[数组](#)

[字符串](#)

[对象和类](#)

[Java的包\(package\)](#)

[访问权限](#)

[抽象类](#)

[接口与回调函数](#)

[容器类和映射类](#)

[线程的创建](#)

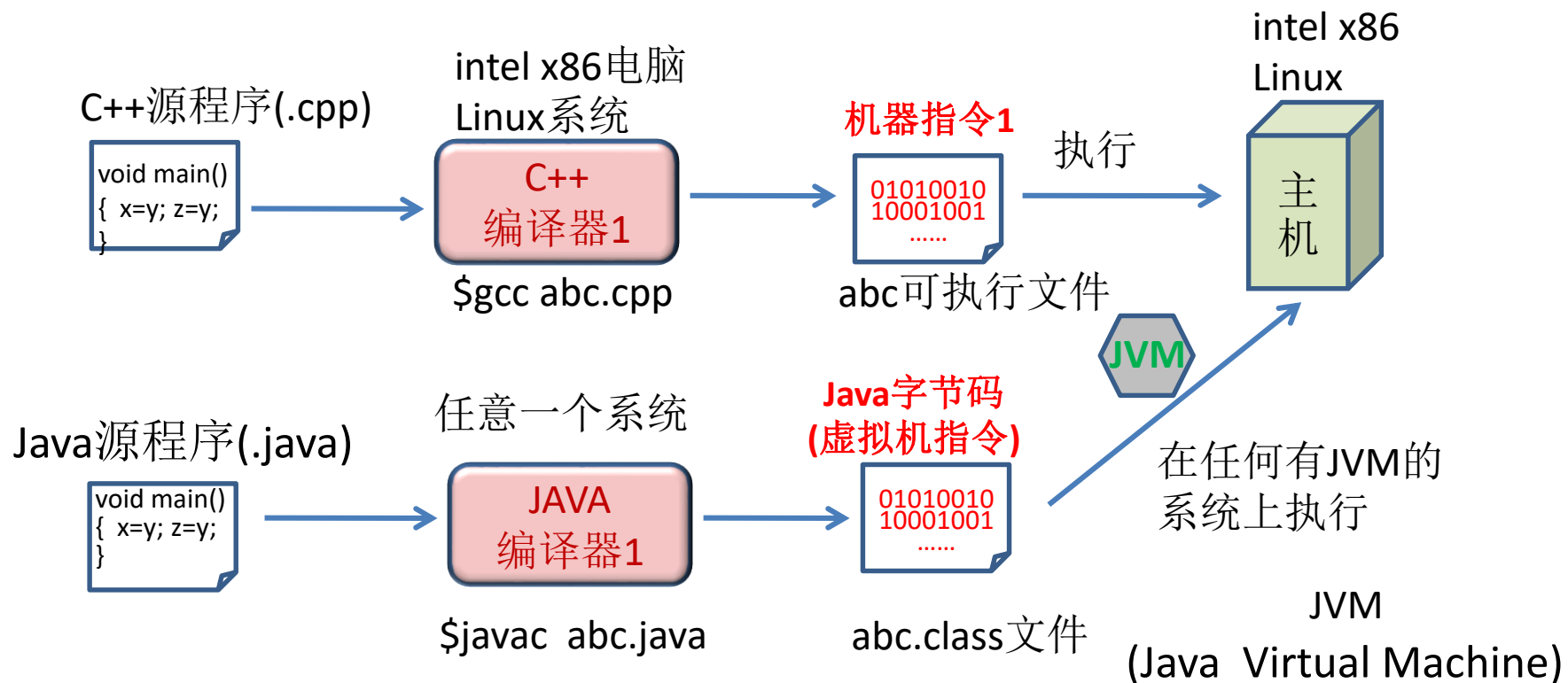
[重载、覆盖与传址](#)

[final关键字](#)

[静态成员](#)

C++编译器和Java虚拟机

- 如果C++源程序(.cpp)想在intel x86电脑的Linux系统上运行，则要用其gcc编译成机器指令文件，该文件只能在intel x86电脑的Linux系统上执行。
- Java源程序(.java) 只要在任何系统上用其Java编译器编译成Java字节码文件(.class)，就可以在任何安装了Java虚拟机(JVM)的系统上执行。



Java的特点

□ 简单(Simple)

设计Java的目的之一就是**简化C++**的功能，使其易于学习和使用。Java没有指针类型(pointer)，避免了内存溢出等安全性问题。Java的垃圾收集器(garage collection)自动释放不再使用的存储空间，使程序员解除了常常忘记释放存储空间的烦恼。

□ 易于移植(Portable)

通过使用Java字节码(byte code)，Java支持交叉平台代码，Java程序可以在任何运行了Java虚拟机的环境中执行，其执行速度被高度优化，其效率有时甚至超过了C++的程序。

□ 面向对象(Object-oriented)

Java语言中一切都是对象，并且Java程序带有用于检查和解决对象访问的运行时(run-time)类型信息。

JRE和JDK

- Java平台有两个主要产品：Java Runtime Environment (JRE) 和Java Development Kit (JDK) 。
- JRE提供Java核心库、Java虚拟机以及运行Java应用程序所需的其它组件。字节码文件就是在JRE下运行的。
- JDK是JRE的超集，用于开发Java小程序和Java应用程序。它包括了JRE所有的内容, 并且加入了编译器和调试器等工具。
- JDK共有三个版本：

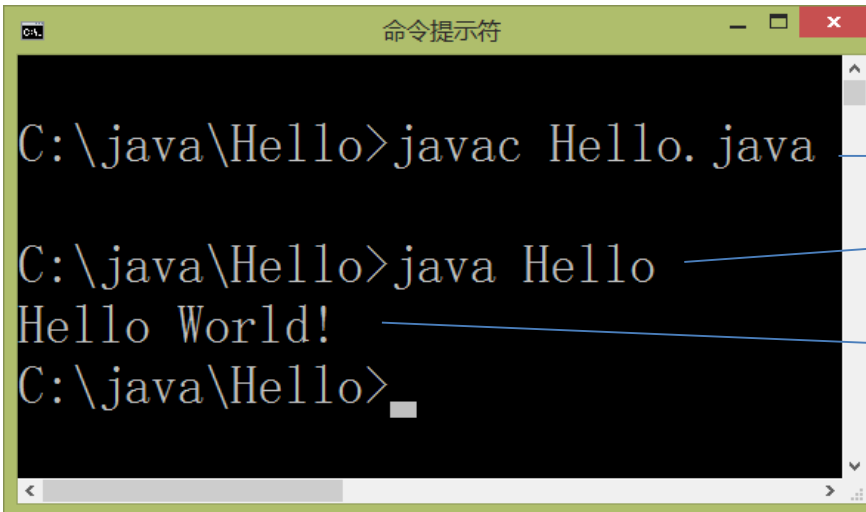
- ✓ **Standard Edition**(Java SE): 标准版，是最常用的一个版本。
- ✓ **Enterprise Edition**(Java EE): 企业版，用于开发大型Java应用程序。
- ✓ **Micro Edition**(Java ME): 微型版，用于移动设备上java应用程序。

* 下载和安装JDK的方法见附录1

第一个Java程序

文件名: Hello.java

```
public class Hello // 类名，要与文件名相同!!!
{
    public static void main(String args[]) // 主程序入口
    {
        System.out.print("Hello World!"); // 显示Hello World!
    }
} // print()为系统对象System.out的方法
```



```
C:\java\Hello>javac Hello.java
C:\java\Hello>java Hello
Hello World!
C:\java\Hello>
```

编译并生成Hello.class

运行Hello.class

显示结果

* `Javac -encoding UTF-8 Hello.java` (UTF-8的源码，默认为ansi编码)

数据类型

- 程序的每个数据都要以一定的格式存放，因此需要定义数据类型，比如，整数类型、字符类型等。
- Java中共有8种基本数据类型：

类型	字节	包装类	数的范围	默认值
byte	1	Byte	-128~127	0
short	2	Short	-32768~32767	0
int	4	Integer	$-2^{31} \sim 2^{31}-1$	0
long	8	Long	$-2^{63} \sim 2^{63}-1$	0
float	4	Float	$3.4e^{-038} \sim 3.4e^{+038}$	0.0
double	8	Double	$1.7e^{-308} \sim 1.7e^{+308}$	0.0
char	2	Character	0~65535	0
boolean	1	Boolean		false

* `int x=0; Integer y=1;` // x是基本数据，y是对象，但是它们的使用方法相同。

* 基本数据类型作为数据域的默认值为上表最后一列。

数组

- Java的数组是一个对象(Arrays类)，其元素是基本数据类型或对象。
- 数组初始化后，如果数组元素为基本数据类型，则自动取默认值，否则取值null。

ArrayDef.java

```
import java.util.Arrays;
int sample[]; // 定义数组对象（未初始化，不能使用）
sample = new int[8]; // 初始化数组，分配8个元素的存储空间。
sample[7]=100; // 数组引用方法
System.out.println(sample[7]); // 显示第7个元素：100。下标从0开始
System.out.println(sample[0]); // 显示：0（默认值）。

int rnds[] = new int[]{1,3,4,5,6};
System.out.println(Arrays.toString(rnds)); //显示：[1,3,4,5,6]
char[] chars = {'我', '是', '中', '大', '人'}; //初始化一维字符数组
System.out.println(chars[3]); //显示字符：大。
String[] s1= {"John", "Wade", "James"}; //初始化一维字符串数组
System.out.println(s1[1]); // 显示字符串：Wade
```



```

int nums[] = {9, -10, 18, -978, 9, 287, 49, 7};
for(int num:nums){                                // 枚举循环法
    System.out.println(num);                      // 显示数组nums的全部元素
}
double map[][] = new double[3][10];              // 定义二维数组：3行10列
map[0][9] = 20;
System.out.println(map[0][9]);

```

// Java只有一维数组，二维数组为数组的数组。 数组的每行的列数是可变的。

```

int table[][] = {{1},{2,3,4},{5,6,7,8}};          // 二维数组（可变长）
for(int i=0; i<table.length; i++){                // table.length为行数
    for(int j=0; j<table[i].length; j++){          // 处理每行的元素
        System.out.println(table[i][j]);          // 显示第i行第j列的元素
    }
}

```

//下面程序做了什么？

```

int table1[][] = new int[10][];
for(int j=0; j<table1.length; j++){
    table1[j]=new int[j+1];
}

```

```
import java.util.Arrays;
```

ArrayOp.java

```
char s1[]={ 'H', 'e', 'l', 'l', 'o' };
s1=Arrays.copyOf(s1,8); // 复制出一个8元素数组:Hello*** (*为null)
System.out.println(s1); // Hello*** (显示为空格)
char s2[];
s2=Arrays.copyOf(s1,3); // 复制: s2得到一个3元素数组:Hel
char s3[]=Arrays.copyOfRange(s1, 1, 3); // 复制: s3得到el

Arrays.fill(s2, 'a'); // 把s2的全部元素填充为a
System.out.println(s2); // 结果:aaa
Arrays.fill(s3,2,5, 'o'); // 把s1的第2~4个元素填充为o
System.out.println(s3); // 结果:Heoo
boolean r = Arrays.equals(s1,s2); //比较元素个数和值是否都相等:false
System.out.println(r);
```

```
int pos=Arrays.binarySearch(s1,'1'); // （二分）查找值为1的元素
System.out.println(pos);
Arrays.sort(s1); // 排序s1:***Hello *为null
System.out.println(s1);
int a[]={3,5,4,26,19,2,9};
Arrays.sort(a,1,5); // 排序第1~4个元素:3,4,5,19,26,2,9
for(int x:a){
    System.out.println(x);
}
```

* `binarySearch()`:使用二分搜索算法来搜索指定的 `int` 型数组，以获得指定的值。**必须在进行此调用之前对数组进行排序**（通过上面的 `sort` 方法）。如果没有对数组进行排序，则结果是不明确的。如果数组包含多个带有指定值的元素，则无法保证找到的是哪一个。

字符串

字符串类型(String)为一个用于文字操作的类，其值为一串字符，采用Unicode编码存储。

```
import java.util.Arrays;
char c1[] = {'a', 'b', 'c', 'd', 'e'};
String s1 = "Hello";
String s2 = new String("World");
String s3 = new String(c1);
String s4 = new String(c1, 1, 3);
String s5[] = {"This", "is", "a", "test."};
String s6 = s1.concat(s2);
String s7 = s1 + s2;
boolean b1 = s1.equals(s2);
boolean b2 = s1.equalsIgnoreCase("hello");
boolean b3 = (s1 == s2);
boolean b4 = s1.isEmpty();
int len = s1.length();
String s8 = Arrays.toString(s5);
```

StringDef.java

```
// 定义一个字符数组
// 建立字符串: Hello
// 建立字符串: World
// 从字符数组建立字符串: abcde
// 从字符数组子集建立字符串: bcd
// 字符串数组
// 字符串并置: HelloWorld
// 字符串并置(同concat): HelloWorld
// s1和s2 (对象内容)是否相同
// 相等比较, 忽略大小写: true
// 是否为同一个对象(变量值)
// 空串判断, 与equals("")相同
// 字符串长度: 5
// 转变为用逗号隔开的
// 字符串: This,is,a,test.
```

StringOp.java

```
import java.util.Arrays;
import java.util.regex.*;

String s1 = "Hello";
char c1[]={ 'W', 'o', 'r', 'l', 'd' };
String s2 = String.copyValueOf(c1);

String s3 = s1.toUpperCase();
String s4 = s1.toLowerCase();
String s5 = s1.substring(0,4);
String s5a= s1.substring(2);
char c2 = s1.charAt(4);
String[] s6 = s1.split("e");
String s7 = s1.replace("l", "L");
String s8 = s1.replaceAll("l", "L");
String s9 = s1.replaceFirst("l", "L");
String s8a = s1.replaceAll("[Hl]", "L");
String s9a = s1.replaceFirst("[Hl]", "L");
String s10 = "    We learn Java";
boolean b1=s10.endsWith("Java");
boolean b2=s10.startsWith("We");
String s11=s10.trim();
```

//把字符数组变为字符串:World

//变大写字母:HELLO

//变小写字母:hello

//第0到3个字符的子串:Hell

//第2个字符开始的子串:llo

//取第4个字符:o.

//分割字符串: "H","llo"

//替换字符串(所有):heLLo。

//替换字符串(所有):heLLo

//替换第一次出现:heLlo

//替换所有字符H或l:LeLLo

//替换第一次出现字符H或l:Lello

// 以什么子串结尾:true

// 以什么子串开始:false

// 删除头尾空格:We learn Java

```

int i=25;
String s12=String.valueOf(i);           // 把整数转换为字符串
boolean b3=s10.contains("learn");       // 是否包含子串learn: true
int pos1 = s11.indexOf("e");             // 第一个匹配子串的索引:1 未找到返回-1.
int pos2 = s11.indexOf("e",3);           // 结果:4 从位置3开始查找
int pos3 = s11.lastIndexOf("e");         // 从尾部开始查找子串:4
s12=String.format("%05d,%s",501,"op");  // 格式化字符串: 00501,loop
String regex =“(;|,)”;                  // 正则表达式 :;或, 与[;,]相同
String[] s14= "a;b,c;d".split(regex);    // 以;或,拆分字:"a","b","c","d"

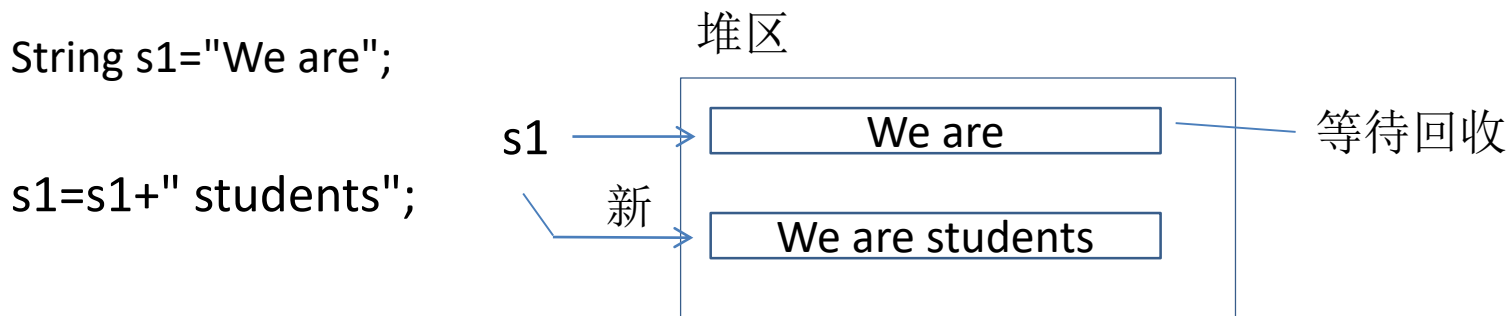
boolean b4=s11.matches("^We.*") ;       // 匹配正则表达式(以We开头):true.
int n1 = "abcd".compareTo("abcD");       // 词典序:32 0等于<0小于>0-大于
int n2 = "abcd".compareToIgnoreCase("abcD"); // 忽略大小写

bytes[] s14 = s11.getBytes("ISO-8859-1"); // 把s11(GB2312)转换成字节数组
String s15 = new String(s14,"GB2312");   // 再把字节数组转换为字符串
int n1=s1.codePointAt(i);                 // 取到s1的第i字符的编码(UNICODE)

```

- * ISO-8859-1编码为单字节编码，常用于把任何编码的字符转换为单字节数组。
- * 数值和日期与字符串之间的转换方法见附录。

String 是不可变的对象，每次修改其内容都会生成了一个新的 String 对象，原来的String对象将不再使用，等待垃圾回收器自动清理它们，因此，频繁改变字符串变量会十分低效。



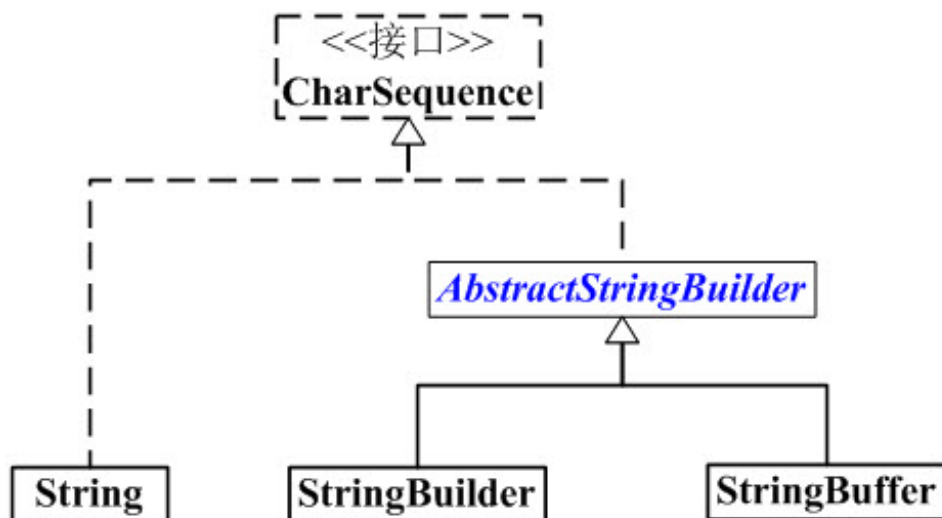
为了提高效率，可以使用**StringBuilder**和StringBuffer。类StringBuffer和StringBuilder (java.lang.*) 的主要操作有 append 和 insert 方法：

```
StringBuilder s20=new StringBuilder("uv");  
s20.append("xyz");           // 并入末尾。uvxyz  
s20.insert(3,"w");           //插入到中间。uvwxyz  
String s21=s20.toString();    //取出s20的内容
```

StringBuffer对方法有同步机制，一般用于多线程环境，而StringBulider适用于单线程环境。对于单线程编程，StringBulider比StringBuffer更有效率。

StringBuffer和StringBuilder对象的常用方法

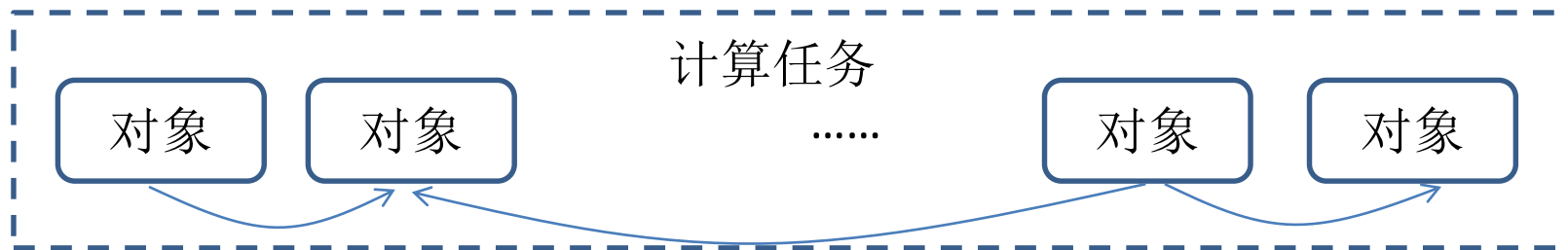
```
append(String str)           // 数值变量等类型也可以作为参数
insert(int offset, String str) // 第二个参数和append一样
delete(int start, int end)    // 删除一个子串
indexOf(String str, int fromIndex) // 从fromIndex开始查找一个子串
replace(int start, int end, String str) // 从start到end替换一个子串
substring(int start, int end)
reverse()                     // 字符反转。"abc"=>"cba"
length()
toString()                    // 取出缓存的字符串
```



- CharSequence是一个操作字符数组的接口，它只包括length(), charAt(int index), subSequence(int start, int end)这三个方法。

对象和类

- 要完成一个计算任务必然要涉及很多事物，在面向对象程序设计中把这些事物称为**对象(object)**，并把具有相同属性和操作的对象划分为一类，定义为类(class)。因此，对象也称为类的实例(instance)。
- 与**面向过程程序设计**采用函数调用完成计算任务不同，**面向对象程序设计**是通过对象之间的功能调用来完成计算任务。



- 在Java中，对象的属性和操作被称为**数据域(data field)**和**方法(method)**，有时也被称为**成员变量(数据成员、字段)**和**成员函数**，统称为对象的成员。
- 面向对象程序设计方法有哪三个主要特征？

(1) 封装性 (2) 继承性 (3) 多态性

封装性(Encapsulation)是指把属性和操作封装为一个整体，使用者不必知道操作实现的细节，只是通过对象提供的操作来使用该对象提供的服务。

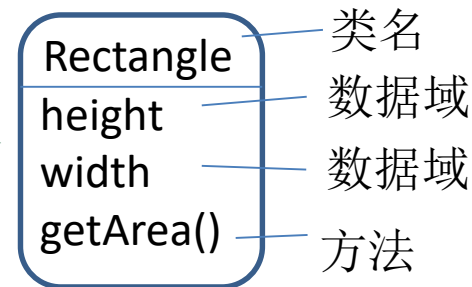
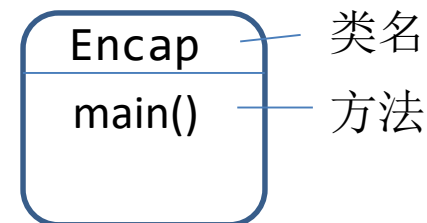
下面是一个计算面积的类的例子，在main()中用其功能时并不需要知道其中的计算方法。

```
class Rectangle {           // 类名
    double height;           // 数据域(成员变量)
    double width;            // 数据域(成员变量)
    double getArea(){        // 方法(成员函数)
        double area=height*width; // 局部变量
        return area;         // 返回值
    };
}
```

```
public class Encap {        // 类名
    public static void main(String args[]){ // 方法
        Rectangle rect = new Rectangle(); // 建一个新对象
        rect.height=10;      // 数据域赋值
        rect.width=20;        // 数据域赋值
        System.out.println(rect.getArea()); // 调用方法
    }
}
```

//运行结果: 200.0

Encap.java



一个java文件可以包含很多class，但是只能有一个public的class。类的信息会保留在其class文件的头部。右边时是Java对一个数据域赋值的标准方法。

```
void setHeight(val){ height=val;}
int getHeight(){return height;}
```

通过**继承性(Inheritance)**，子类（导出类、派生类、继承类）可以自动共享父类（基类、超类）的属性和操作。子类与父类之间是**is-a**（完全继承）或**is-like-a**（加入了自己的属性或方法）的关系。**Java**的所有基类自动继承内部的**Object**类。

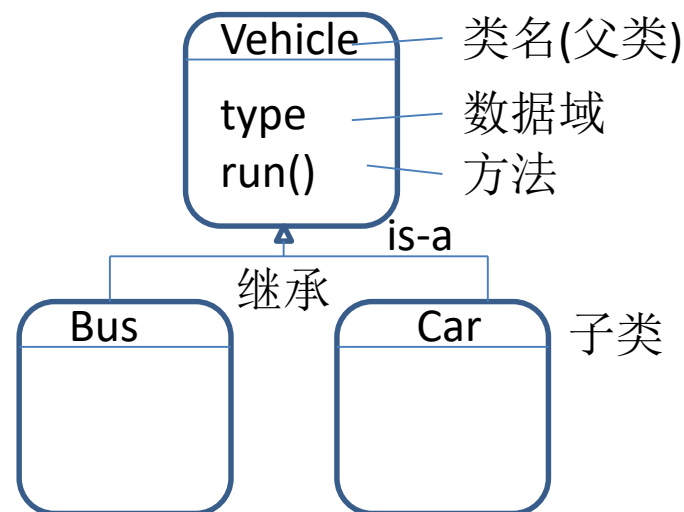
下面是继承性的例子：

```
class Vehicle{                                //父类(基类)名
    int type;                                //数据域
    void drive(){                             //方法
        System.out.println("run!" + this.getClass());
    };
}
class Bus extends Vehicle{ //子类(导出类)
}
class Car extends Vehicle{ //子类名
}
```

```
public class Inherit{
    public static void main(String args[]){
        Bus bus = new Bus();
        bus.drive();
        Car car = new Car();
        car.drive();
    }
}
```

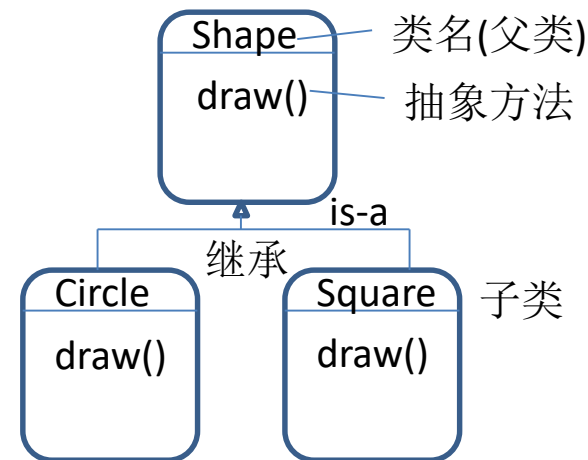
```
//运行结果: run! class Bus
//          run! class Car
```

祖先类
子孙类



多态性(Polymorphism)是指对同一类的不同对象采用相同的操作时可以产生完全不同的行为。下面是多态性的例子：

```
class Shape{           // 父类(基类)
    int color;         // 数据域
    void draw(){       // 方法
        System.out.println("draw! "+this.getClass());
    }
}
class Circle extends Shape{ // 子类(导出类)
    void draw(){
        System.out.println("draw!"+this.getClass());
        /* super.draw() // 可以用于访问父类中的方法 */
    };
}
class Square extends Shape{
    @Override          // 覆盖(override)才是多态
    void draw(){ System.out.println("draw! "+this.getClass());
    };
}
```



```
public class Poly{
    public static void main(String args[]){
        Shape shape1 = new Shape(); shape1.draw();
        Shape shape2 = new Square(); shape2.draw();
        Shape shape3 = new Circle(); shape3.draw();
    }
}
```

运行结果：

```
draw! class Shape
draw! class Square
draw! class Circle
```

* Square和Circle对象都向上转型为Shape类。

* 向上转型都是后期绑定，即运行时绑定。前期绑定是编译时的绑定。

* 调用泛化类（父类）对象的方法，产生的行为是由具体类（子类）决定的，因此呈现多态特性。

Java的包(package)

[参考](#)

• 概念

- 一个大的Java工程会包含很多class文件。Java通过子目录对工程所用的class文件进行分类管理。
- 同一个子目录下的所有.class文件被认为在同一个包（package）中。还可以把多个包放在同一个.jar文件中。

根目录/
|
子目录com
|
子目录group
|
子目录food
|
包com.group.food的所有.class文件

• 定义包

- 使用网站名(group.com)来确保包名的唯一性。
- 没有定义包的文件属于默认包。

Java源文件--定义包

```
package com.group.food  第一行  
.....
```

• 引用包

- Java通过import语句指出类所在的包来引用类。
- 没有指出包的类属于默认包。
- 直接引用包可以处理不同包的同名类：
Cookie cookie = new com.group.food.Cookie();

Java源文件--引用包

```
import com.group.food.*;  
Cookie cookie = new Cookie();
```

```
import com.group.food.Cookie;
```

• 使用包的例子

- Cookie.class和Bread.class放在目录c:\java\PackEx\com\group\food下

```
package com.group.food;
public class Cookie {
    public void eat(){
        System.out.println("Eat cookie");
    }
}
// Cookie.java
```

```
package com.group.food;
public class Bread {
    public void eat(){
        System.out.print("Eat bread");
    }
}
// Bread.java
```

- PackEx.java放在目录c:\java\PackEx下,编译时会自动编译包的类

```
import com.group.food.*; // PackEx.java
public class PackEx{
    public static void main(String[] args){
        Cookie cookie = new Cookie();
        cookie.eat();
        Bread bread = new Bread();
        bread.eat();
    }
}
// 运行结果: Eat cookie
//           Eat bread
```

如果Cookie.java和Bread.java都放入包com.group.food中,编译PackEx.java时会自动编译它们。

采用环境变量CLASSPATH指明包的查找路径,例如, set CLASSPATH =c:\java\PackEx;c:\classes;. 定义了三个查找路径(.为当前路径)。

• 使用jar文件的例子

第一步、把.class文件放在子目录(com\group\food)中。

```
package com.group.food;
public class Corn {
    public void eat(){
        System.out.print("Eat corn");
    }
}
```

Corn.java

```
package com.group.food;
public class Rice {
    public void eat(){
        System.out.println("Eat rice");
    }
}
```

Rice.java

第二步、执行命令"jar cvf food.jar *" 得到文件food.jar （内部包含了子目录信息）

第三步、把food.jar放在c:\java\PackJar\中，加入c:\java\PackJar\food.jar到CLASSPATH。

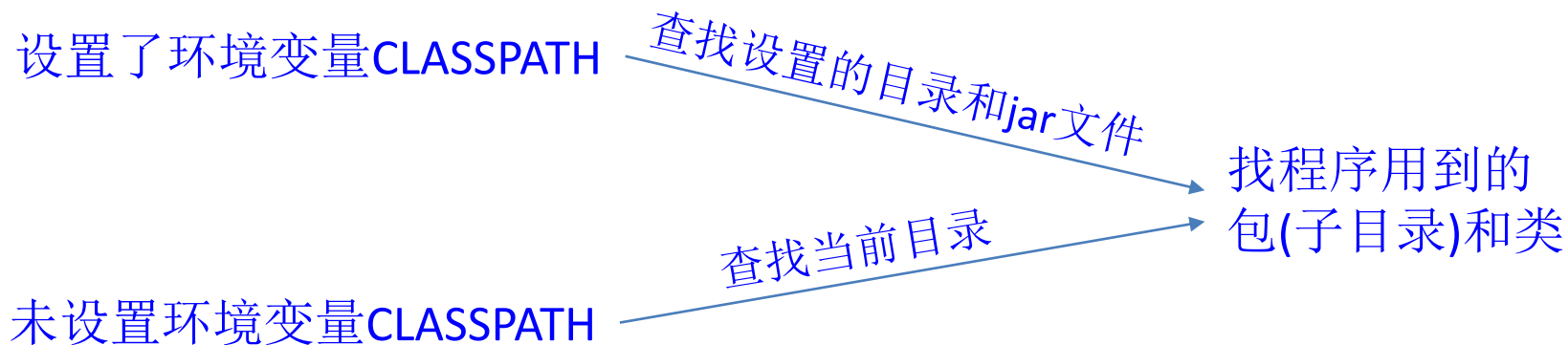
```
import com.group.food.*;
public class PackJar{
    public static void main(String[] args){
        Rice rice = new Rice();
        rice.eat();
        Corn corn = new Corn();
        corn.eat();
        Bread bread = new Bread();
        bread.eat();
    }
}
```

PackJar.java

```
set CLASSPATH
=c:\java\PackEx;c:\classes;.;
c:\java\PackJar\food.jar
```

```
// 运行结果:  Eat Rice
//              Eat Corn
//              Eat bread
```

• JVM如何找到所需要的包和类？



- Java目录lib下jar也在查找之列。
- 没定义包的class文件被看成使用默认包。
- 默认包只能放在CLASSPATH设置的目录下，没有设置CLASSPATH，则放在当前目录下。
- 如果出现多个同名的包，只会使用第一个出现的同名包。
- 编译和运行时可以用参数-CLASSPATH或-cp直接指出CLASSPATH:
`javac -cp c:\java\PackJar\food.jar;c:\java;. PackJar.java`
`java -cp c:\java\PackJar\food.jar;c:\java;. PackJar`
- CLASSPATH或cp的路径中不要使用环境变量和~，但是可以用其它环境变量。
- Linux下的CLASSPATH使用冒号“:”间隔目录而不是用分号“;”

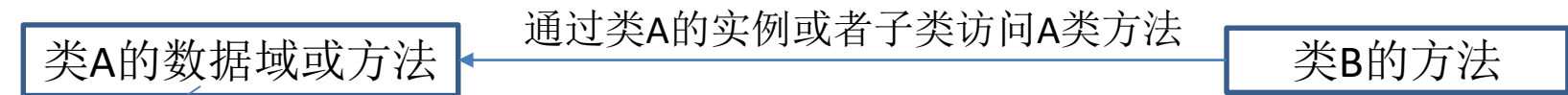
访问权限

• 类的访问权限

没有修饰词 只能被同一个包的类所访问
public 能被任何类在任何地点所访问

* 内部类有**private**和**protected**权限，还可以定义为**static**的。

• 数据域和方法的访问权限



public 能被任何类在任何地点所访问
protected 只能被导出类和同一个包的类所访问
private 只能被同一个类的方法所访问
无修饰词 只能被同一个包的类所访问
(类似friendly)

	pub	pro	pri	无修饰词
同一个类	√	√	√	√
子类（非同包）	√	√		
同包类（含子类）	√	√		√

总结：对于不同包之间，只能访问**public**的或者父类**protected**的数据域和方法。对于相同包之间，可以访问除了**private**的所有其他权限的数据域和方法。

抽象类

一个类如果有方法声明没有方法主体（body），则要把该方法定义为抽象方法。具有抽象方法的类必须定义为抽象类。**抽象类不能被用于创建实例**，但是可以用于定义子类。

```
abstract class ShapeAbs {                                // 抽象类
    public ShapeAbs() {                                    // 构造函数(constructor)
        System.out.println("Shape Initialized!");
    }
    public abstract void draw();                          // 抽象方法(method)
}
class Circle extends ShapeAbs {
    public void draw() {                                    // 定义方法draw()
        System.out.println("CircleA draw() is called!");
    }
}
public class ShapeInherit {
    public static void main(String args[]){
        Circle circle1 = new Circle();
        circle1.draw();
    }
}
```

如果一个类的某个方法只能由子类给出具体实现，就可以定义为抽象类。例如：只有知道具体是什么动物才知道它是怎么移动的。

接口与回调函数

```
interface Door {  
    void open();  
    void close();  
}  
  
class MyDoor implements Door {  
    @Override  
    public void open(){ //回调函数  
        System.out.println("open door!");  
    }  
    @Override  
    public void close(){ //回调函数  
        System.out.println("close door!");  
    };  
}  
  
public class IntfExample1 {  
    static void enterRoom(Door door){  
        door.open(); door.close();  
    }  
    public static void main(String[] args){  
        Door door = new MyDoor(); // 向上转型  
        enterRoom(door);  
    }  
}
```

创建接口的匿名子类的实例的例子：

```
interface Door {  
    void open();  
    void close();  
}  
  
public class IntfExample2 {  
    static void enterRoom(Door door){  
        door.open(); door.close();  
    }  
    public static void main(String[] args){  
        Door door= new Door(){           // 创建一个Door的子类的实例  
            @Override  
            public void open(){ //回调函数  
                System.out.println("open door!");  
            }  
            @Override  
            public void close(){ //回调函数  
                System.out.println("close door!");  
            }  
        };  
        enterRoom(door);  
    }  
}
```

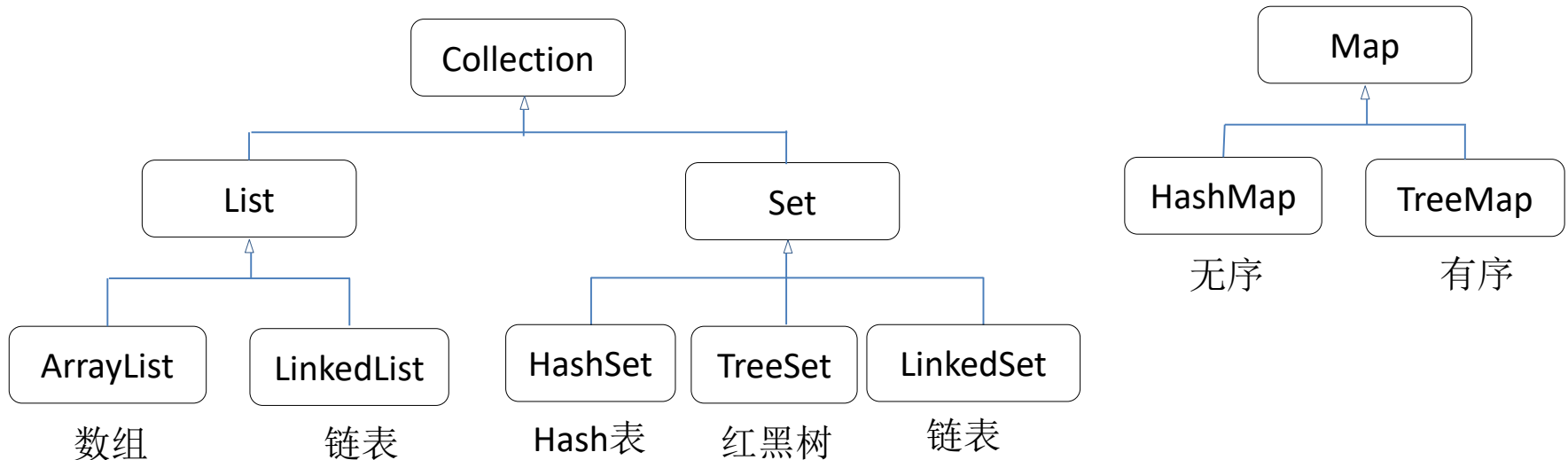
把接口的匿名子类的实例作为参数的例子：

```
interface Door {
    void open();
    void close();
}

public class IntfExample3 {
    static void enterRoom(Door door){
        door.open(); door.close();
    }
    public static void main(String[] args){
        enterRoom(new Door(){        // 创建一个Door的子类的实例
            @Override
            public void open(){ //回调函数
                System.out.println("open door!");
            }
            @Override
            public void close(){ //回调函数
                System.out.println("close door!");
            }
        });
    }
}
```

容器类和映射类

- 数组对最大的元素个数有限制，不能进行插入和删除，查找元素速度也很慢，使用容器类和映射类可以解决这些问题。
- 容器类**Collection**可以存放需要经常增加元素或删除元素的一组元素，其主要子类为**Set**(无重复无序)和**List**(有序可重复)。
- 映射类**Map**可以将键(Key)映射到值(Value)。其中的键值不能重复，每个键值只映射到一个值。



* 容器类还有Vector类（用于多线程环境）、Stack类(Vector的子类)、Queue类。

* 映射类还有Dictionary类(已过时)，Hashtable类(已过时，Dictionary的子类)。

• ArrayList和LinkedList

ArrayList是一个用顺序存储结构实现线性表的类，随机访问速度快，插入删除操作比较慢，而LinkedList用链表实现，插入删除快，随机访问速度慢。

```
import java.util.*;
public class ArrayListTest {
    public static void main(String[] args) {
        ArrayList<Student> stus = new ArrayList<Student>(); // 泛型(参见附录)
        Student stu1 = new Student(101, "Wang");
        Student stu2 = new Student(102, "Li");
        Student stu3 = new Student(103, "He");
        Student stu4 = new Student(112, "李四");
        stus.add(stu1); // 尾部追加
        stus.add(stu2); stus.add(stu3);
        stus.add(1, stu4); // 中间插入
        Iterator<Student> it = stus.iterator();
        while (it.hasNext()) { // 顺序取出
            System.out.println("***" + it.next());
        }
        System.out.println("+++" + stus.get(2)); // 随机取出
        stus.forEach(stu->{stu.name=stu.name+"*"; System.out.println(stu)});
        List<Student> stus2=Arrays.asList(stu1,stu2,stu3); // 用数组转，不能add
        System.out.println("###" + stus2);
    }
}
```

执行结果

```
***101 Wang
***112 李四
***102 Li
***103 He
+++102 Li
101 Wang*
112 李四*
102 Li*
103 He*
###[101 Wang*, 102 Li*, 103 He*]
```

* stus可以用List声明，向上转型。-> lamda

```

class Student {
    int num;           // 学号
    String name;       // 姓名
    Student(int num, String name) { this.num = num; this.name = name; }
    public String toString() { return num + " " + name; }
}

```

ArrayList和LinkedList的主要方法：泛型E可以使用用户自定义类或标准类(Integer, String等)

```

boolean add(E e)           // 将指定的元素添加到列表尾部。E为泛型
void add(int index, E e)   // 将指定的元素插入列表指定位置。
boolean addAll(List list)  // 将列表所有元素插入到当前列表中，返回正确或错误。
void clear()              // 移除列表的所有元素。
boolean contains(Object o) // 如果列表包含指定的元素，则返回 true。
E get(int index)          // 返回列表指定位置上的元素。
int indexOf(Object o)     // 返回列表首次出现的指定元素的索引或 -1。
boolean isEmpty()         // 如果列表为空，则返回 true
int lastIndexOf(Object o) // 返回列表最后一次出现指定元素的索引或 -1。
E remove(int index)       // 移除此列表中指定位置上的元素。
boolean remove(Object o)  // 移除列表中首次出现的指定元素（如果存在）。
E set(int index, E e)     // 用指定的元素替代此列表中指定位置上的元素。
int size()               // 返回此列表中的元素数。
void trimToSize()         // 将此 ArrayList 实例的容量调整为列表的当前大小
protected void removeRange(int fromIndex, int toIndex)
                          // 移除 fromIndex和 toIndex(不包括)之间的所有元素。

```

* 在多线程环境下使用CopyOnWriteArrayList代替ArrayList，链式使用ConcurrentLinkedQueue。

* ConcurrentLinkedQueue能使用上述不带index的方法和前面的Iterator循环。

• HashMap和TreeMap

随机访问和插入删除操作很快，HashMap要更快一些，但是内存占用量更大。TreeMap的键值可以有序遍历，而HashMap不行。

```
import java.util.*;
public class HashMapTest {
    public static void main(String[] args) {
        Map<Integer, Student> stus = new HashMap<Integer, Student>();
        Student stu1 = new Student(103, "He");
        Student stu2 = new Student(112, "李四");
        Student stu3 = new Student(101, "Wang");
        stus.put(103, stu1);    // 加入新的键值对
        stus.put(112, stu2);
        stus.put(101, stu3);
        Iterator<Integer> it = stus.keySet().iterator();
        while (it.hasNext()) { // 无序遍历
            Integer key = (Integer) it.next();
            Student value = stus.get(key); // 根据键值取出值
            System.out.println("***" + value.toString());
        }
        System.out.println("+++" + stus.get(112));
    }
}
```

* stus向上转型为Map类，也可以继续使用HashMap。

```
***112 李四
***101 Wang
***103 He
+++112 李四
```

运行
结果

HashMap的主要方法:

void clear()	// 从此映射中移除所有映射关系。
Object clone()	// 返回此 HashMap 实例的副本(不复制键和值)
boolean containsKey(Object key)	// 是否包含指定键
boolean containsValue(Object value)	// 是否包含指定值
V get(Object key)	// 返回指定键所映射的值或null。
boolean isEmpty()	// 映射集是否为空。
Set<K> keySet()	// 返回此映射中所包含的键的 Set 视图。
V put(K key, V value)	// 在此映射中关联指定值与指定键。
V remove(Object key)	// 从此映射中移除指定键的映射关系(如果存在)
int size()	// 返回此映射中的键-值映射关系数。
Collection<V> values()	// 返回此映射所包含的值的 Collection 视图。

- * 泛型K使用标准类(Integer, String等)
- * 泛型V还可以使用用户自定义类
- * 在多线程环境下使用**ConcurrentHashMap**

线程的创建

- Thread类是接口Runnable的子类， Runnable接口具有回调函数run()。通过创建Thread子类的实例可以创建线程。启动该线程时执行该实例的回调函数run()。

```
Thread thread = new Thread() {public void run() {} };  
thread.start();
```

- 还可以通过直接创建Thread实例来创建线程，此时需要把Runnable接口的子类的实例作为其构造器的参数。启动该线程实际上就是执行该Runnable实例中定义的回调函数run()。

```
Runnable runnable = new Runnable(){public void run() {} };  
Thread thread = new Thread(runnable);  
thread.start();
```

例1、通过创建Thread子类的实例创建线程:

```
class MyThread1 extends Thread {
    static int cnt = 0;
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            cnt++;
        }
        System.out.println("cnt="+cnt);
    }
}

public class ThreadTest1 {
    public static void main(String[] args) {
        Thread thread= new MyThread1();
        thread.start();
    }
}
```

* 加入构造器可以给出线程名MyThread1(String thr){ **super(thr);** }

例2、通过创建Thread的匿名子类的实例创建线程：

```
public class ThreadTest2 {  
    public static void main(String[] args) {  
        new Thread(){  
            int cnt = 0;  
            @Override  
            public void run() {  
                for (int i = 0; i < 10; i++) {  
                    cnt++;  
                }  
                System.out.println("cnt="+cnt);  
            }  
        }.start();  
    }  
}
```

* 用new Thread(String thr)可以带入线程名

例3、利用Runnable接口的实例来创建线程：

```
class MyRunnable implements Runnable {
    Integer cnt = 0;
    @Override
    public void run() {
        for (int i = 0; i <= 10; i++) {
            cnt++;
        }
        System.out.printf("cnt="+cnt);
    }
}

public class ThreadTest3 {
    public static void main(String[] args) {
        Runnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

- 可以自定义线程名： `new Thread(runnable, "thd-1")`
- main的三条语句可以简写为 `new Thread(new MyRunnable()).start;`

例4、通过创建Runnable接口的匿名子类的实例创建线程：

```
class ThreadTest4 extends Thread{
    static int cnt = 0;
    public static void main(String[] args) {
        new Thread(new Runnable(){
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    cnt++;
                }
                System.out.println("cnt="+cnt);
            }
        }).start();
    }
}
```

* new Runnable(){ 得到接口Runnable的匿名子类的实例。

例5、在实现Runnable接口的子类中创建线程:

```
public class ThreadTest5 {  
    public static void main(String[] args) {  
        ThreadTest5A test = new ThreadTest5A();  
        test.start();  
    }  
}
```

```
class ThreadTest5A implements Runnable {  
    Integer cnt = 0;  
    public void start() { // 在主线程执行  
        Thread thread = new Thread(this);  
        thread.start();  
    }  
    @Override  
    public void run() { // 在子线程执行  
        for (int i = 0; i < 10; i++) {  
            cnt++;  
        }  
        System.out.printf("cnt="+cnt);  
    }  
}
```


例6、在实现Runnable接口的主类中创建线程：

```
public class ThreadTest6 implements Runnable {
    Integer cnt = 0;
    public static void main(String[] args) {        // 在主线程执行
        Thread thread = new Thread(new ThreadTest6());
        thread.start();
    }
    @Override
    public void run() {                                // 在子线程执行
        for (int i = 0; i < 10; i++) {
            cnt++;
        }
        System.out.printf("cnt="+cnt);
    }
}
```

重载、覆盖与传址

- 一个类中同时定义参数个数不同或参数类型不同的多个同名方法的做法叫**重载(Overload)**。
- 子类也可以重载父类的方法，但是如果参数完全相同，则会覆盖(**Override**)父类的方法，使父类的同名方法不能直接使用，要通过super对象进行调用。
- 不能根据返回值不同类型来定义重载。
- 在方法前加上**@Override**可以让编译器检查父类是否有这个方法，没有则会出错。
- Java没有传址参数，要想用参数带返回值需要使用**自定义类**作为参数来实现。
- 定义为**final**的方法不能被覆盖。

```
class A {  
    int x1;  
    void do(){ x1=5;}  
    void do(int y){ x1=y;} overload  
}  
class B extends A {  
    @Override  
    void do(int y){  
        super.do(y); override  
        .....  
    }  
}
```

final关键字

```
final int COUNT = 20;  
private static Random rand = new Random();  
static final int INT_5 = rand.nextInt(COUNT);    //0-19  
final StringBuilder sb = new StringBuilder("Hello");  
sb.append("12345");
```

- 定义为**final**的数据域和局部变量在初始化后不能被再次赋值，如果它们是引用型的，其内容(对象)可以被修改。**final**的局部变量必须在定义时进行初始化，而**final**的数据域可以在定义之后进行初始化，例如，在构造器中初始化。
- **final**参数在方法内不能被再次赋值。
- 定义为**final**的方法不能在子类中被覆盖(Override)。**private**方法会被隐式地指定为**final**方法。
- 定义为**final**的类不能被用于定义子类。
- 如果采用字面量直接初始化（还可以调用函数进行初始化），**final**的数据域和局部变量被Java编译器当成**常量**。

静态成员

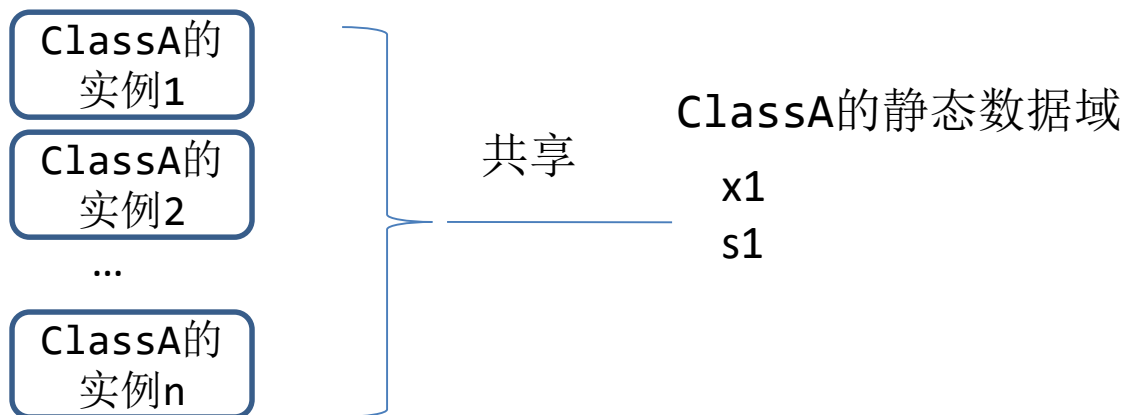
静态数据域和静态方法不需要建立实例就可以直接访问。实际上在它们第一次被访问时系统会装载它们所在的类同时将它们初始化，直到程序结束它们才被释放。

```
class ClassA {  
    static int x1 = 0;  
    static String s1;  
    {  
        ClassA.s1 = "abcd"; // 不会执行（非静态块，未加static）  
    }  
    static void f1(){  
        System.out.println("Hello!");  
    }  
}  
class TestStatic {  
    public static void main(String[] args){  
        ClassA.f1(); // 访问静态方法 Hello!  
        System.out.println(ClassA.x1); // 访问静态属性 0  
        System.out.println(ClassA.s1); // null  
        TestStatic t = new TestStatic();  
        t.f(12); // 访问非静态方法 12  
    }  
    public void f(int x){  
        System.out.println(x);  
    }  
}
```

只有静态块才会在类装载时被执行，非静态块
只有在创建ClassA实例时才会被执行。

静态方法只有创建实例才能访问同一个类的非静态数据域和方法。

- Java的静态数据域位于内存的方法区，并被所有实例所共享。静态数据域只在装载它所在的类时分配内存并初始化一次。



- Java的静态数据域和方法可以看成存在于整个程序执行期，因此，它们可以作为类似C++的全局变量和函数进行使用。与C++不同，Java没有静态局部变量。
- 静态方法中不能使用**this**和**super**，构造器不允许被声明为**static**，这是因为**this**和**super**、构造器都是与实例相关的。
- 静态方法不能被非静态方法覆盖，也不能直接访问非静态数据域或方法，而是要通过建立实例进行访问。非静态方法可以访问静态数据域或方法。
- import语句后加入**static**会令该包中的类的所有方法变为**static**。

```
import static com.group.show.*;
```