Java程序设计

(中)

2016.11.19

isszym　sysu.edu.cn

内容

- 可变参数
- 重载、覆盖与传址
- 构造器
- 内部类
- 抽象类和接口
- 捕捉错误
- 作用域和生存期
- Final关键字
- 内存分配
- 静态成员
- 垃圾回收

可变参数

如果希望方法带入不同个数的参数，就可以使用可变参数。

```
class VaryParm{
  static void print(int x, String...  args){
    System.out.println(""+x+"***"+args.length);//参数个数
    for(String temp:args)
       System.out.println(temp);
   }
   public static void main(String[] args){
       print(1);
       print(2, "66","77");
       print(3,"81","82","83");
       String a[]={"a","b","c","d"};
       print(4,a);
   }
}
```

结果：

```
1***0
2***2
66
77
3***3
81
82
83
4***4
a
b
c
d
```

可变参数只能作为最后一个参数使用。 args.length为可变参数的个数，args[i]为第i个可变参数。

http://blog.csdn.net/testcs_dn/article/details/38920323

重载、覆盖与传址

- 一个类中定义多个同名方法的做法叫**重载**(overload)。这些方法主要通过参数个数或类型的不同进行区分。不能用返回值类型不同来重载。
- 子类也可以重载父类的方法。如果参数完全相同，则会覆盖(override)子类的方法，使父类的同名方法不能使用。
- 在覆盖方法前加上@Override，编译器会根据这个指示进行检查是否父类有这个相同的方法。
- 父类中的被覆盖的方法可以通过super.method()调用。
- Java没有指针类型，基本数据类型和字符串类型的参数都不能作为传址参数，要想用参数带回值只能使用**自定义的类**的参数实现。

```
class A {
    int x1;
    void do(){
        x1=5;
    }
    void do(int y){
        x1=y;
    }
}
```

构造器

```
class BaseA {
   int x1;
   BaseA(){
       x1=5;
   }
   BaseA(int y){
      x1=y;
   }
}
class DerivedA extends BaseA{
   String s1;
   DerivedA(String s2,int y){
      super(y);
      this.s1="'"+s2+"'";
   }
}
class TestConst {
     public static void main(String[] args){
        DerivedA o1= new DerivedA("",12);
        System.out.println(o1.s1+o1.x1);

     }
}
```

构造器可以重载，默认的访问类型为public

- 默认的构造器
- 自动调用父类的默认或无参数构造器(如果有)。
- 如果调用父类有参数构造器，则不会在调用其它构造器。

- 构造器(constructor)主要用于初始化对象中的成员变量，可以带或不带参数，其名字要与其类名完全相同。
- 构造器默认的访问类型为public，如果定义为private，则不能在外部直接建立对象，而需要通过调用其静态方法建立对象。
- 构造器可以重载，每个类可以定义多个构造器，对于多个构造器，系统要根据新建对象时使用的参数情况来选择不同的构造器。
- 如果一个类没有构造器，Java系统会自动为它生成一个**默认的构造器**。
- 当建立子类对象时，要先**调用父类的构造器**。如果父类没有构造器，子类会先自动调用父类的默认构造器，否则，会自动调用没有参数的构造器。父类的有参数构造器不会被子类自动调用，需要在子类构造器中用super(参数)调用，此时不会再调用其他构造器了。
- 如果基类只有带参数的构造器而子类又没有调用它，则子类会调用基类的默认构造器。

内部类

在类的内部定义的类为内部类。内部类可以被内部方法所使用，并对外部实现了隐藏。在内部类中可以直接访问其它外部类。

```
class InClass {
     int x=0;
     class B {
        void f1(){
          x = 5;
        }
     }
     B getB(){
        return new B();
     }
     public static void main(String[] args){
         InClass a1= new InClass();
         System.out.println(""+a1.x); //输出0
         B b1 = a1.getB();
         b1.f1();
         System.out.println(""+a1.x);  //输出5
     }
}
```

内部类还可以定义在类的方法内部。

抽象类和接口

- 如果一个方法只有声明而不定义方法主体，则要把该方法定义为抽象方法。带有抽象方法的类必须定义为抽象类。抽象类不能被用于定义对象。
- 当父类的方法不愿或者不能使用默认行为时，这个方法最好被定义为抽象方法。

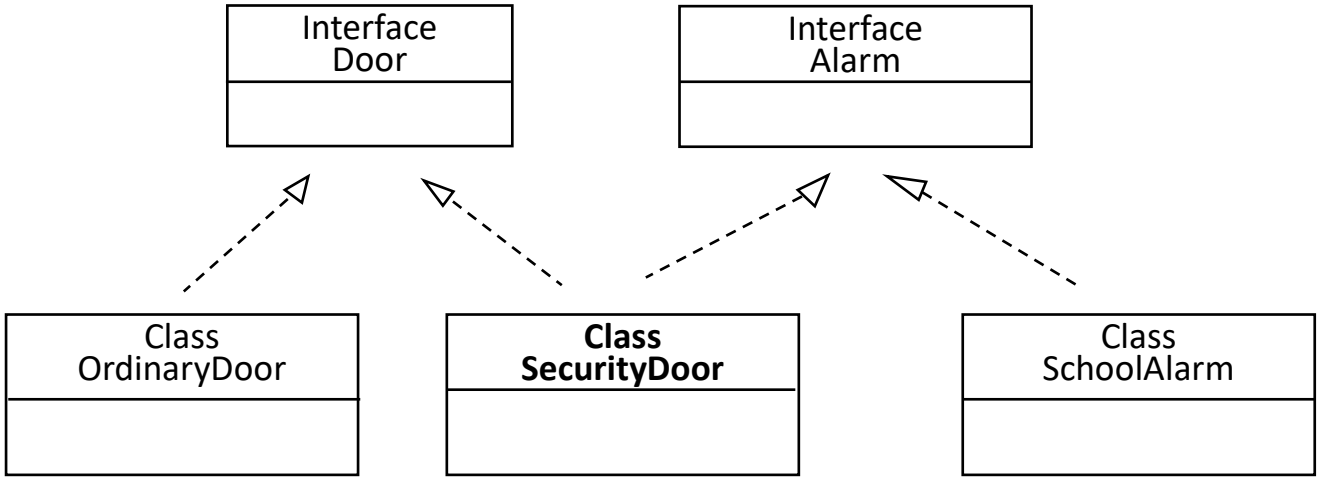
```
abstract class ShapeAbs {                       // 抽象类
   String color;                                // 变量或属性(field)
   public ShapeAbs() {                          // 构造函数(constructor)
      System.out.println("Shape Initialized!");
      color = "black";
   }
   public abstract void draw();                 // 抽象函数或方法(method)

   public void setColor(String color) {         //方法: 设置颜色
      this.color = color;                       //this.color表示本类的属性
   }

   public String getColor() {                   //方法: 取出颜色
      return this.color;
   }
}
```

```
class CircleA extends ShapeAbs {
   public CircleA() {        // 构造函数(constructor)
      System.out.println("CircleA Initialized!");
   }
   public void draw() {          // 定义方法draw()
      System.out.println("CircleA draw() is called!");
   }
}
class RectangleA extends ShapeAbs {
   public void draw() {
      System.out.println("RectangleA draw() is called!");
   }
}
public class ShapeInherit {
 public static void main(String args[]){
      CircleA circle1 = new CircleA();
      drawShape(circle1);
      RectangleA rectangle1 = new RectangleA();
      drawShape(rectangle1);
   }
   static void drawShape(ShapeAbs shape){
      shape.draw();
   }
}
```

执行结果：

```
Shape Initialized!
CircleA Initialized!
CircleA draw() is called!
Shape Initialized!
RectangleA draw() is called!
```

与C++不同，Java没有多重继承，Java的多重继承可以通过实现多个接口来做到。




```
public class SecurityDoor implements Door,Alarm {
    public void open(){
        System.out.println("open door!");
    }

    public void close(){
        System.out.println("close door!");
    }

    public void alarm(){
        System.out.println("alarm!");
    }
}
```

如果采用接口来搭建上层模块，只要接口不变，即接口的方法声明不变，增加该接口的子类、增加接口的方法以及修改子类的方法体都不会引起上层模块的改变。

```
public class InterfaceTest {
   public static void main(String args[]){
      OrdinaryDoor door1 = new OrdinaryDoor();
      SchoolAlarm alarm1 = new SchoolAlarm();
      SecurityDoor door2 = new SecurityDoor();
      enter(door1);
      enter(door2);
      alarm(alarm1);
      alarm(door2);
   }
  public static void enter(Door door){
     door.open();
     door.close();
  }
  public static void alarm(Alarm alarm){
     alarm.alarm();
  }
}
```

一个类除了可以实现多个接口，还可以同时继承一个基类。例如：下面ClassA继承ClassB，实现了接口InterfaceA和InterfaceB。

源码见附录

```
class SlidingDoor {
   void slidingOpen(){
      System.out.println("open door by sliding!");
   }
   void slidingClose(){
      System.out.println("close door by sliding!");
   }
}
class SlidingSecurityDoor extends SlidingDoor implements Door,Alarm  {
      //除了继承SlidingDoor的方法，还要实现Door和Alarm的方法
}
```

可以通过继承多个接口定义更大的接口:

```
public interface SecurityDoorInterface extends Door,Alarm {
    public void radio(); //除了继承Door和Alarm的，还增加了radio()的声明
}
```

捕捉错误

```
public class CatchError {
   public static void main(String args[]){
      int x;
      try {                                    // 打开例外处理语句
         for (int i = 5; i >= -2; i--) {
            x = 12 / i;                        // 出现例外(x==0)后将不执行后面的语句
            System.out.println("x=" + x);// 直接跳到catch内执行
         }
      }
      catch (Exception e) {                  // 捕捉例外信息。可以并列用多个catch
         System.out.println("Error:"+e.getMessage()); // 显示当前错误信息
         // e.printStackTrace();             // 显示系统错误信息
      }
      finally{                               // 出现例外必须执行这里的语句
          x=0;
      }
      System.out.println(x);

   }
}
```

```
执行结果:
x=2
x=3
x=4
x=6
x=12
Error:/ by zero
0
```

如果一个方法不愿意处理一些例外，可以采用throws定义方法的例外，把这些例外交给调用者去处理。

```
public class DivideClass {
    void divide() throws Exception {       // 出错后交给调用程序处理
      for (int i = 5; i >= -2; i--) {
         int x = 12 / i;                   // 出现例外(x==0)后将不执行后面的语句
         System.out.println("x=" + x);
      }
    }
}
public class ThrowError  {
   public static void main(String args[]){     // main为主程序入口
      try {                                     // 打开例外处理语句
           DivideClass div = new DivideClass();
           div.divide();
      }
      catch (Exception e) {                     // 捕捉例外信息。可以并列用多个catch
         System.out.println("Error:"+e.getMessage()); // 显示当前错误信息
      }
   }
}
```

执行结果：

```
x=2
x=3
x=4
x=6
x=12
Error:/ by zero
```

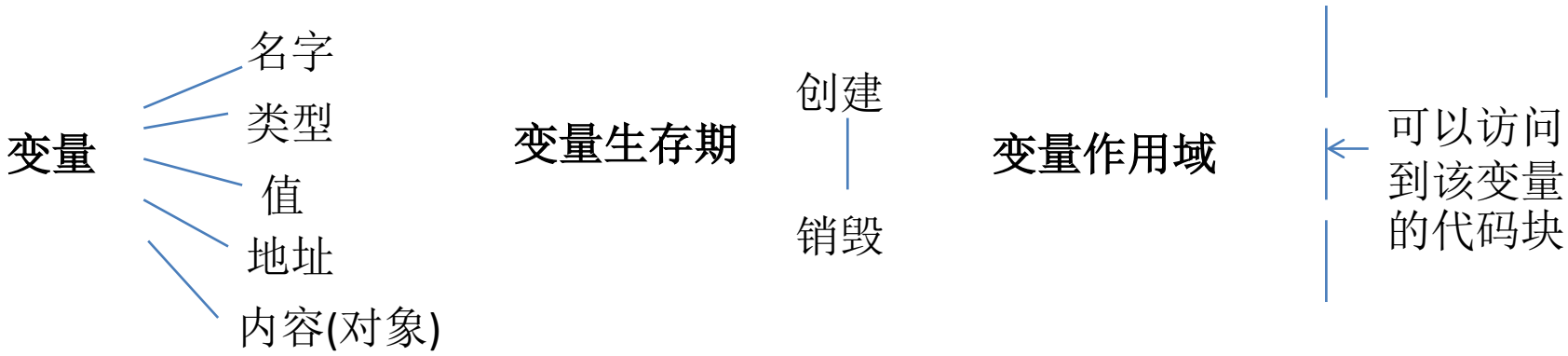
作用域和生存期

```
class  MyMath {
  String msg;
  int sum(int n){
    int res=0;
    for(int i=1;i<n;i++){
       n+=i;
    }
    return res;
  }
}
```

数据域

局部变量的作用域：定义它的块
局部变量的生存期：它所在的方法被调用的时候

Java没有全局变量！



- 变量的类型指出值的类型或引用类型，变量的地址指出值存放的地址。对象变量为**引用类型的变量**，其值指向对象的存放位置。

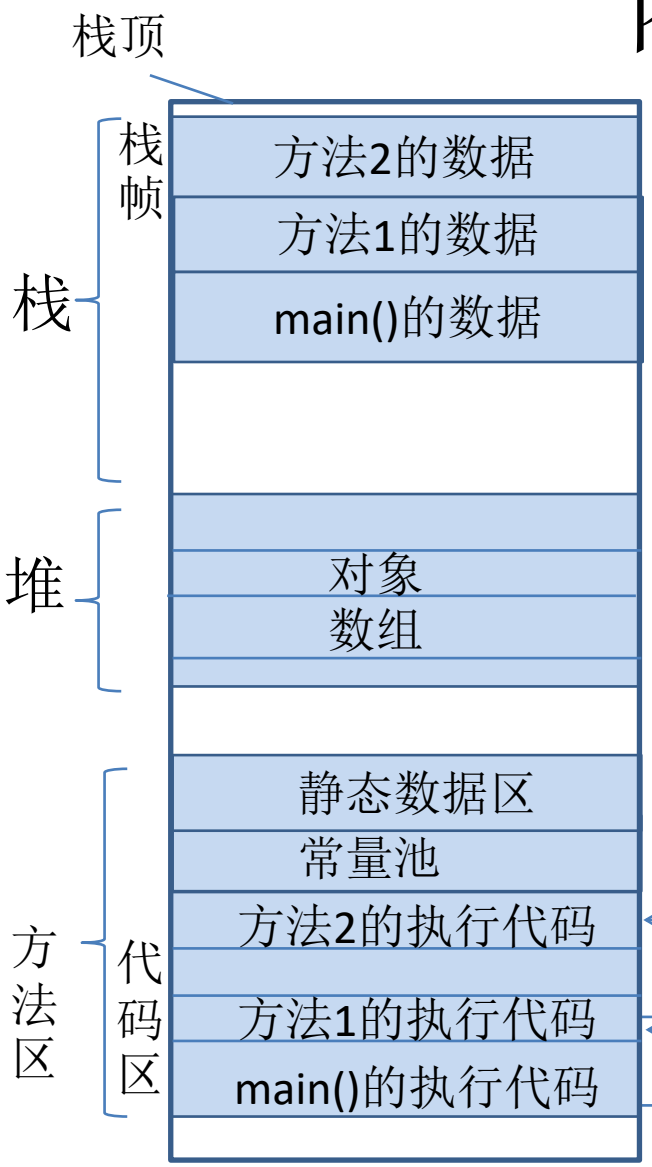
- 何时可以访问变量取决于变量的生存期和作用域。**变量的作用域**是指可以访问到该变量的代码范围。**变量的生存期**是指变量占用内存的时间，如果超出变量的生存期，即使在作用域内也不能访问该变量。
- **局部变量的生存期**为从其所在方法被调用到退出方法。**非静态数据域(成员变量)的生存期**与其所在对象的生存期相同。
- **所有局部变量的作用域**都局限于定义它的块(由{}括住)。Java的同名局部变量不能在某个块及其子块中同时定义（C语言可以，只使用最近定义的）。**非静态成员变量的作用域**由其所在对象的访问权限确定。
- 没有初始化的数据域的取值：基本数据类型的数据域取值为0、0.0或者false，类数据域为null。未初始化的局部变量均为未定义，不能直接使用。

Final关键字

```
private static Random rand = new Random(47);
static final int INT_5 = rand.nextInt(20);     //0-19
```

- 定义为final的数据域、局部变量或者参数在初始化后不能被修改。
- 定义为final的方法不能被子类所覆盖(override)。
- 定义为final的类不能被继承。
- 定义为final的数据域的初始化可以在定义之后进行，但是只能初始化一次。

内存分配



每个线程：PC(程序计数器)

JVM(方法)栈帧：

每次调用方法时进栈

局部变量(值或对象引用)

方法参数

方法返回值

操作数栈（计算表达式）

常量引用

本地方法栈(调用C的动态链接库)

静态数据域

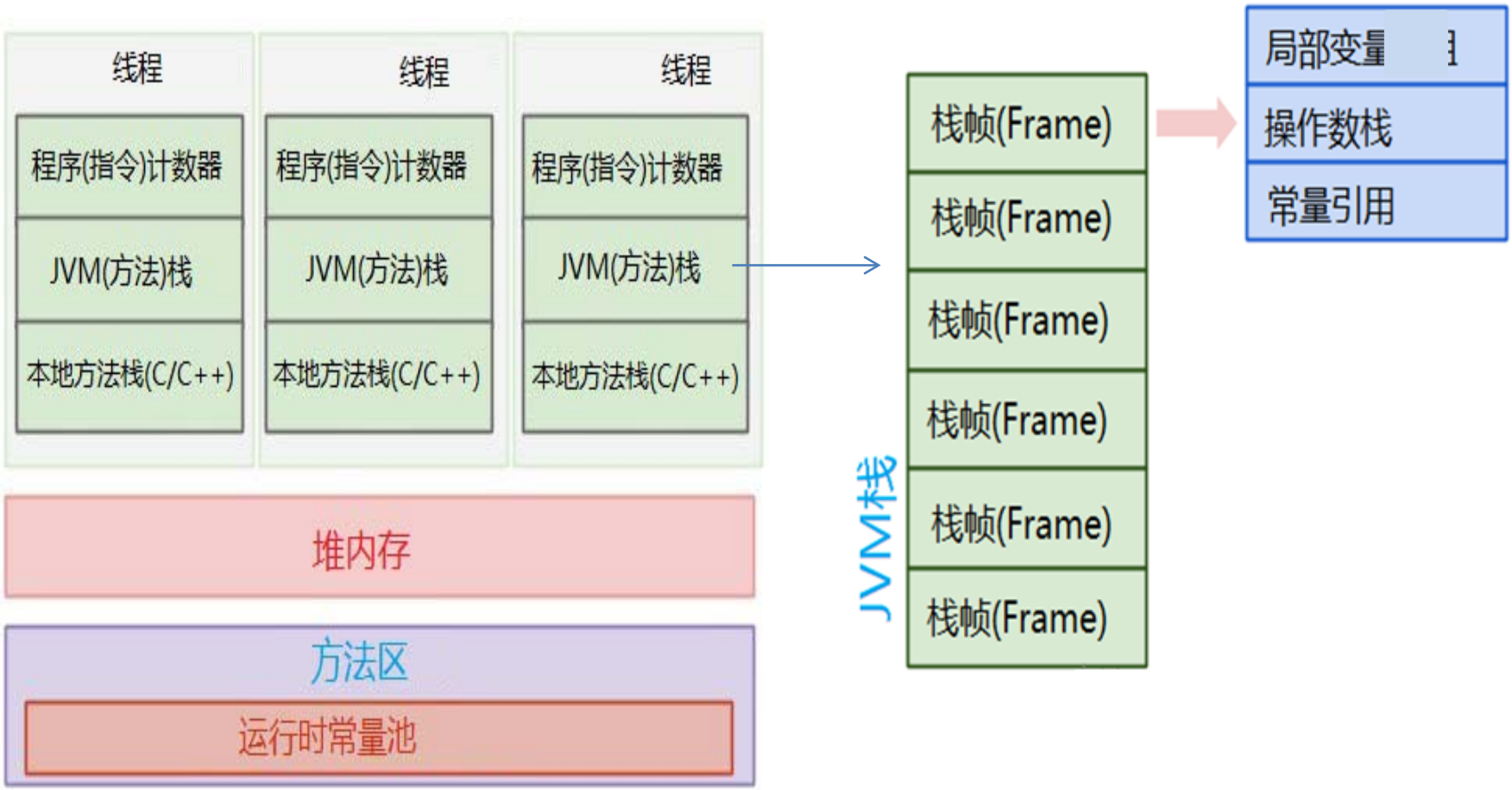
存放常量、类、接口等的信息

调用

调用

代码区包含类的构造器和方法

* 定义对象变量只是定义了对象的引用（在栈中）只有new之后才在堆中为对象分配内存。


线程
线程
线程
程序(指令)计数器
程序(指令)计数器
程序(指令)计数器
JVM(方法)栈
JVM(方法)栈
JVM(方法)栈
本地方法栈(C/C++)
本地方法栈(C/C++)
本地方法栈(C/C++)
堆内存
方法区
运行时常量池
JVM栈
栈帧(Frame)
栈帧(Frame)
栈帧(Frame)
栈帧(Frame)
栈帧(Frame)
栈帧(Frame)
局部变量
操作数栈
常量引用

栈(Stack)区：

每个线程都有PC(程序计数器)、JVM(方法)栈帧以及本地方法栈(调用C语言等的动态链接库用)。每个栈帧存放用于一个方法的局部变量、方法参数、方法返回值、操作数栈（计算表达式）、常量引用。对于基本数据类型变量，直接存放值；对于对象变量，只存放对象引用 。由于编译器可以直接确定局部变量等的相对地址(相对栈帧基址)，所以访问速度很快。

堆(heap)区：

用于存放数组和对象。堆中对象当没有任何引用时，其空间通过垃圾回收进行释放。每一个Java应用都有一个JVM实例，每一个实例都有自己的堆。

方法区：

包含静态数据区、常量池和代码区。代码区保存类的方法和构造器的代码。

常量池的常量分为字面量和引用量。文本字符串、final变量等都是字面量，类信息、接口信息、数据段信息、方法信息都属于引用量。引用量最常见的一种用法是在调用方法的时候，根据方法名找到方法的引用，并以此定位到方法体进行执行。

JVM把常量池按数据类型存放常量，并通过索引访问它们的入口。放在常量池的内容在编译期间就被确定，并被保存在已编译的.class文件中。

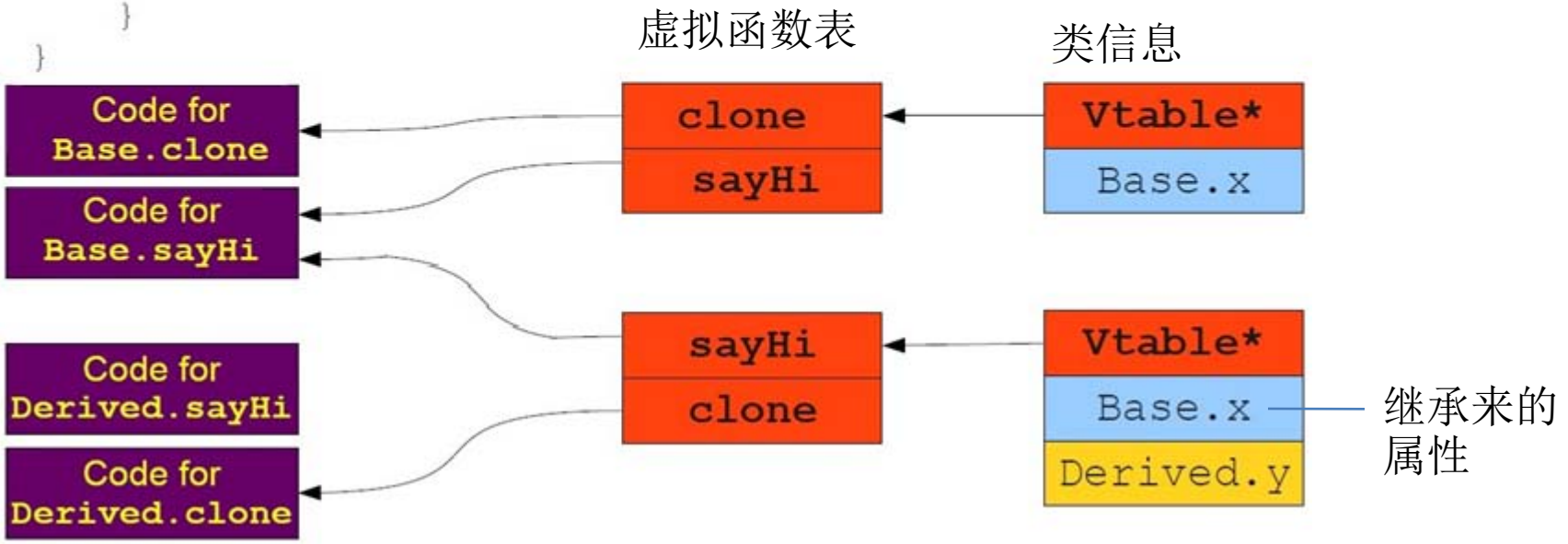
Java编译器会自动对常量进行优化，会查找并使用常量池已有的常量，见下例：

```
String str1 = new String("abc");
String str2 = new String("abc");
String str3 = "abc";
String str4 = "abc";
String str5 = "ab"+"c";
System.out.println(str1 == str2);     //false
System.out.println(str1 == str3);     //false
System.out.println(str3 == str4);     //true
System.out.println(str4 == str5);     //true
```

Java对象在内存中是怎样分配的呢？新建对象时JVM要在常量池中找到放在Class对象中的类信息，然后在堆中建立对象，再把地址填入对象变量中。一旦对象在堆中分配了空间，那本质上就是一块连续内存字节，那么如何找到对象的某个特定属性域和方法的起始地址呢？虚拟机是通过常量池的类信息来找到的。

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base {
    int y;
    Derived clone() {
        return new Derived;
    }
}
```



每个类有一个虚拟函数表，保存了该类的所有方法，包括继承来的方法。虚拟函数表的每个表项会指向实际的方法代码块的起始位置。

静态成员

Java的静态数据域和静态方法都是在第一次被访问时在堆中分配内存空间，然后一直保留到程序结束才释放。对它们的引用都是直接放在访问它们的指令中。

```
class ClassA {
    static  int  x1 =0;
    static String s1;
    {
        ClassA.s1 = "abcd";  //不会被执行!
    }
    static void f1(){
        System.out.println("Hello!");
    }
}
class TestStatic {
     public static void main(String[] args){
          ClassA.f1();
          System.out.println(ClassA.x1);
          System.out.println(ClassA.s1);
     }
     public void f(int x){
          System.out.println(x);
     }
}
```

显示：

```
Hello
0
null

12
```

```
TestStatic t = new TestStatic();
t.f(12);
```

- 静态成员(指数据域和方法)不需要建立实例就可以直接访问，非静态成员只有在建立实例后才能被访问。
- 实例方法可以访问静态成员和实例成员，但是静态方法只允许访问静态成员。
- 静态成员都是在第一次被访问时分配内存空间的，所占空间一直使用到程序结束才被释放。静态数据域被所有实例所共享。静态方法中不能使用this（因为this是与实例相关的）。
- main()就是静态方法，它也不用预先建立对象就可以让系统直接调用。Java的静态数据域和静态方法可以作为类似C++的全局变量和函数进行使用。Java没有静态局部变量。
- import语句后加入static可以令该包中的类的所有方法变为static。

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

垃圾回收

- Java语言没有析构器(destructor)，不能主动销毁对象，所有对象都由垃圾回收器自动进行回收。
- 如果一个对象如果没有任何引用，则可以被回收。
- Java的垃圾回收器只有在内存缺乏时才会去销毁这些没有任何引用的对象，并触发对象的finalize事件。因此，回收工作也许在程序结束都不会发生。
- 可以把对象的变量赋值为null来删除变量对该对象的引用。对于局部变量，在所在方法退出后系统会自动清除引用。
- 如果需要在对象使用结束前做一下清理工作，例如，关闭文件，把一些引用类变量设置为null，可以自己在对象中专门定义一个方法来处理。

附录1 继承类和实现接口

```
interface Door {
          void open();   // 开门
          void close();  // 关门
}
class OrdinaryDoor implements Door {
          public void open(){ System.out.println("open door!");}
          public void close(){ System.out.println("close door!");}
}
interface Alarm {
           void alarm();   // 拉响警报
}
class SchoolAlarm implements Alarm {
          public void alarm(){System.out.println("alarm!");
          }
}
class SecurityDoor implements Door,Alarm {
   public void open(){
      System.out.println("open door!");
   }
   public void close(){
      System.out.println("close door!");
   }
   public void alarm(){
      System.out.println("alarm!");
   }
}
```

```
class SlidingSecurityDoor extends SlidingDoor implements Door,Alarm {
   public void open(){
      slidingOpen();
   }
   public void close(){
      slidingClose();
   }
   public void alarm(){
      System.out.println("alarm!");
   }
}
public class InterfaceTest {
   public static void main(String args[]){
      OrdinaryDoor door1 = new OrdinaryDoor();
      SchoolAlarm alarm1 = new SchoolAlarm();
      SecurityDoor door2 = new SecurityDoor();
      SlidingSecurityDoor door3 = new SlidingSecurityDoor();
      enter(door1);
      enter(door2);
      enter(door3);
      alarm(alarm1);
      alarm(door2);
      alarm(door3);
   }
  public static void enter(Door door){
     door.open();
     door.close();
  }
  public static void alarm(Alarm alarm){
     alarm.alarm();
  }
}
```