# Java类和对象

## 面向对象程序设计

面向对象程序设计（OOP Object Oriented Programming）是当前主流的程序设计架构。它已经逐渐取代了20世纪70年代面向过程的程序设计方式。Java是面向对象的，必须熟悉OOP才能进行java的程序设计与开发。

目前Java，C#，C++等都是面向对象的语言。

C语言是结构化的编程语言(Structured Porgramming SP)。强调精确的数据结构和优越的算法，关注程序的运行效率。程序模块之间的协调性差，程序的重用性，安全性，健壮性较差。难于维护，不适合做较大的应用。

面向对象的程序是由对象组成的。每个对象包含对用户公开的特定功能部分和隐藏的实现部分。程序的对象可以来自标准库，可以来自第三方库，也可以自己定义。

SP和OOP的对比如下图：

|  |  |  |
| --- | --- | --- |
|  | **SP** | **OOP** |
| **设计思路** | 自顶向下，层次化，逐级分解 | 自底向上，对象化，综合 |
| **程序单元** | 函数模块 | 对象 |
| **设计方法** | 程序=数据结构+算法 | 程序=对象=数据+方法 |
| **优点** | 相互独立，代码共享 | 接近人的思维方式，模拟现实世界 |
| **缺点** | 数据与程序不一致，维护困难 | 客观世界无序性，概念不成熟 |

面向对象四大特点：抽象、封装、继承、多态。

* **抽象（Abstraction）：**

忽略一个主题中与当前目标无关的方面，以便更多、更充分的关注与当前目标有关的方面。

抽象就是将一类实体的共同特性抽取出来。重新进行封装或者说封装到新的类中。抽象是面向对象语言的基础。比如说：我们可以通过哈士奇，松狮，金毛抽象出一个叫做狗的类。

里面会包含我们感兴趣的数据，比如说颜色、毛发、耳朵；包含狗的一些我们关注的行为，比如汪汪叫，愉快的跑等。

抽象的作用可以让我们来屏蔽事物的复杂性。有利于我们关注事物的核心部件。

* **继承（Inheritance）：**

继承是一种连接类的层次模型，继承的目的在于鼓励重用。它提供了一种表述共性的方式。

继承用来表示is-a的关系，即特殊类（子类）和一般类（父类）的关系。

* **封装（Encapsulatation）：**

封装是把过程和数据包围起来。对数据的访问只能通过已定义的界面。

封装又叫信息隐藏，是与对象有关的一个概念。将数据和行为隐藏起来。实现封装的关键是绝对不能让类的方法直接访问其他类的实例域，程序只能通过对象的方法与对象的数据进行交互。

成员隐藏访问修饰符权限：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **同一个类** | **同一个包** | **不同包的子类** | **不同包的非子类** |
| **private** | **√** |  |  |  |
| **default** | **√** | **√** |  |  |
| **protected** | **√** | **√** | **√** |  |
| **public** | **√** | **√** | **√** | **√** |

* **多态（Polymorphism）：**

表述一般类中定义的属性或服务被特殊类继承后，可以具有不同的数据类型或表现出不同的行为。通常使用重载（overload）和重写（overwrite）两项技术来实现。

一般有四种方式的实现：

|  |  |  |
| --- | --- | --- |
| **多态类型** | **描述** | **示例** |
| 重载（专用多态） | 描述一个函数名称有不同的实现方式，通常可以在编译时基于类签名来区分各个重载函数的名称。 | class Overload{  public void test();  public void test(int x);  public void test(int x,String y);  } |
| 重写（包含多态） | 是重载的一种特殊情况，只发生在父类和子类关系中。通常签名相同，内容不同。 | class p {  public void test(int);  }  class c extends p {  public void test(int);  } |
| 多态变量（赋值多态强制多态） | 声明为一种类型，但实际上却可以指向另一种数据类型的变量。 | Parent p = new Child(); |
| 泛型（模板，参数多态） | 提供了一种创建通用工具的方法，可以在特定的场合将其特化。 | template<class T> T max(T a,T b){  If(a<b) return b;  Return a;  } |

## 对象和类

### 对象

对象是面向对象编程的核心部分。是实际存在的具体实体，具有明确定义的状态和行为。从本质上来讲对象就是数据和方法的组合体。其中：数据表示自身的状态，也称为属性或成员数据，成员变量等；方法用来表示自身的功能。

在面向对象的设计架构中，世间的万事万物都是对象。比如马云是一个对象，包含的数据有姓名，年龄，生日，职务；可能的行为有发表心灵鸡汤演讲，管理阿里巴巴公司，融资等等。我们自己使用的电脑是一个对象，包含型号，颜色，cpu型号等。可能的行为有浏览网页，看片，炒股等的。

对象的三个主要特征如下：

* **行为（behavior）**

可以对对象施加哪些操作，或者说可以对对象施加哪些方法。

* **状态（state）**

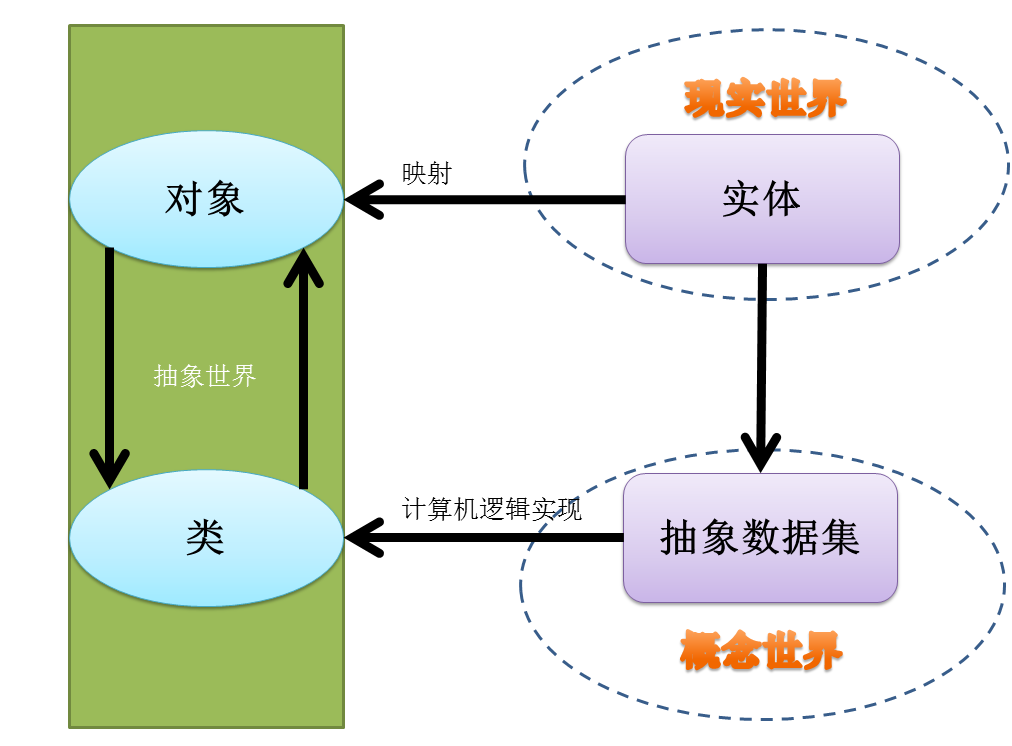
施加方法时对象如何响应。

* **标识（identity）**

如何辨别具有相同状态与行为的不同对象。

### 类

类是构造对象的模板（template）和蓝图（blueprint）。是对现实世界中对象的抽象。



我们可以将类理解为做月饼的模子，要什么形状的月饼制作什么样的模子。不同的模子做出来的月饼就是我们不同模子的一个个的对象。

类是相同属性和行为的一组对象的集合：

* **属性**

事物的特性在类中用变量表示。

每个对象的每个属性都用于其特定的值。

属性名称由类的所有对象共享。

对象或实体拥有的特征在类中表示时称为属性。

* **方法**

事物的行为和动作在类中用函数表示。

每个对象都有相同的动作和行为。

对象执行的操作在类中表示为方法。

类和对象的区别：

* 1. 类是用来描述实体的“模板”或者原型；
  2. 对象是实际的实体，每一个对象都是类的一个具体实例。
  3. 类用来定义对象所有的属性和方法，同一类的所有对象都拥有相同的特征和操作；
  4. 可以将类理解成生成产品的模具，而对象则是根据此模具生产的一个个产品。

### 类之间的关系

#### 依赖（uses-a）

依赖（dependence），即“uses-a”关系，是一种最明显的、最常见的关系。依赖关系表现为一个类（A）的方法操作了另外一个类（B）的对象，我们就说类A依赖于类B。

应该尽可能地将相互依赖的类减至最少。如果类A不知道B的存在，那么B的变化对Ａ就没有影响。这就是类之间的低耦合。

#### 聚合（has-a）

聚合（aggregation），即“has-a”关系，是一种具体且易于理解的关系。聚合的表现为A的对象包含B的对象。

#### 继承（is-a）

继承（inheritance），即“is-a”关系，是一种用来表示特殊与一般的关系。

## 对象和类的使用

### 使用预定义类

在java中没有类，我们什么事情也做不了。所有的活动都是在类中展开的。并不是所有的类都具有面向对象的特征，如Math类，Math类由一系列的方法组成，没有实例数据。

#### 对象与对象变量

要想使用对象，就必须先构造对象。并指定其初始状态，然后对对象使用方法。

在java编程语言中使用构造器（constructor）构造新实例。构造器是一种特殊的java方法，用来构造并初始化对象。

构造器（构造方法）与类名相同，我们来看看java.util.Date类的部分构造方法。

|  |
| --- |
| **public** Date() {  **this**(System.*currentTimeMillis*());  }  **public** Date(**long** date) {  fastTime = date;  }  @Deprecated  **public** ~~Date~~(**int** year, **int** month, **int** date) {  ~~this~~(year, month, date, 0, 0, 0);  }  @Deprecated  **public** ~~Date~~(**int** year, **int** month, **int** date, **int** hrs, **int** min) {  ~~this~~(year, month, date, hrs, min, 0);  }  @Deprecated  **public** ~~Date~~(**int** year, **int** month, **int** date, **int** hrs, **int** min, **int** sec) {  **int** y = year + 1900;  // month is 0-based. So we have to normalize month to support Long.MAX\_VALUE.  **if** (month >= 12) {  y += month / 12;  month %= 12;  } **else** **if** (month < 0) {  y += CalendarUtils.*floorDivide*(month, 12);  month = CalendarUtils.*mod*(month, 12);  }  BaseCalendar cal = *getCalendarSystem*(y);  cdate = (BaseCalendar.Date) cal.newCalendarDate(TimeZone.*getDefaultRef*());  cdate.setNormalizedDate(y, month + 1, date).setTimeOfDay(hrs, min, sec, 0);  getTimeImpl();  cdate = **null**;  }  @Deprecated  **public** ~~Date~~(String s) {  **this**(~~parse~~(s));  } |

接下来以Date为例来学习构建对象的方法。

要想构建一个Date对象需要使用new操作符，如下所示：

new Date()

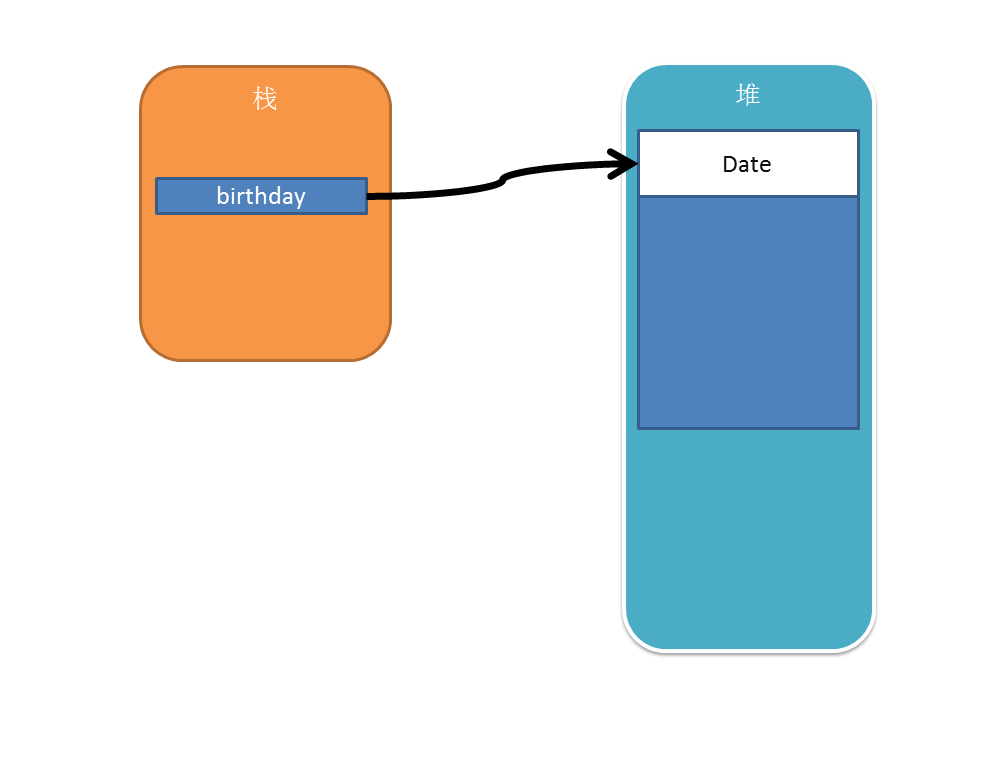
这个表达式构造了一个新对象，这个对象初始化为当前的日期和时间。

|  |
| --- |
| **import** java.util.Date;  **public** **class** DateTest {  **public** **static** **void** main(String[] args){  System.***out***.println(**new** Date());  }  }  打印结果如下：  Tue Jun 28 14:23:32 CST 2016 |

如果我们希望我们构造的对象多次使用，可以将对象保持到一个变量中：

Date birthdayDate = new Date()

该表达式执行后的内存逻辑结构如下：



Date birthdayDate；

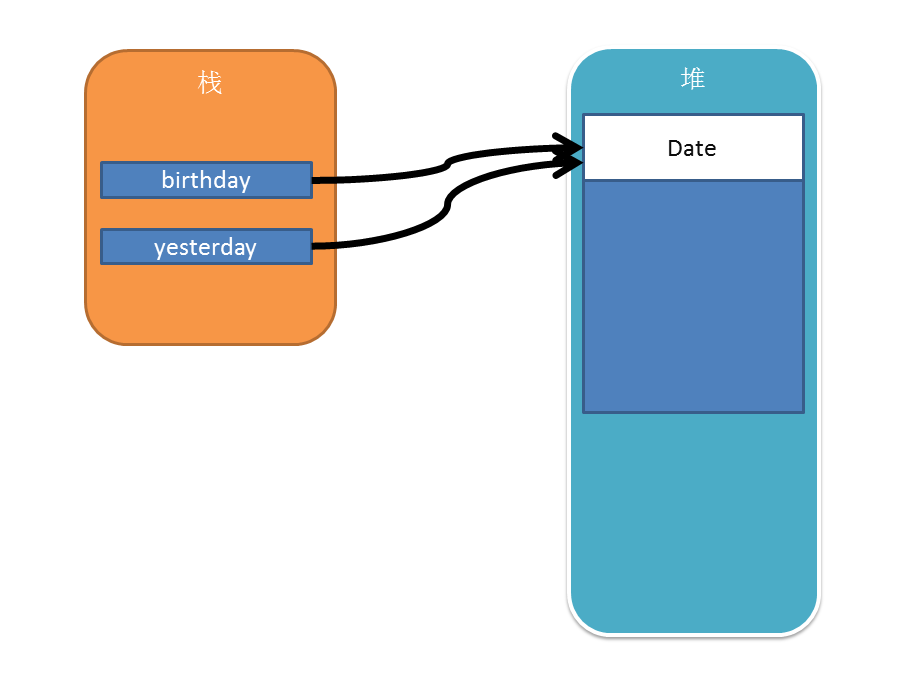
该表达式的意思是定义了一个名称为birthdayDate的变量，它可以引用Date类型的对象。目前情况下birthdayDate不是一个对象，也没有引用任何对象。反应到内存层面是在jvm内存栈中分配了变量本身保存所有要的空间。此时不能对该变量进行任何的方法调用。如：birthdayDate.toString()。

必须首先初始birthdayDate变量才能使用，可以有两种方式初始化：

birthdayDate = new Date()

或者

birthdayDate = yesterday；//yesterday已经初始化



注意：

对象变量本身不包含一个对象，仅仅是一个引用，引用存储在其他位置（堆）的对象。

#### 更改器与访问器（getter，setter方法）

面向对象的一个显著特点就是封装。封装对类的使用者隐藏了实现细节，所有对类数据的修改都要通过类提供的方法。这里说到的getter，setter方法就是其中的一种，我们还是先看看Date提供的一些getter，setter方法：

|  |
| --- |
| @Deprecated  **public** **int** ~~getMonth~~() {  **return** normalize().getMonth() - 1; // adjust 1-based to 0-based  }  @Deprecated  **public** **void** ~~setMonth~~(**int** month) {  **int** y = 0;  **if** (month >= 12) {  y = month / 12;  month %= 12;  } **else** **if** (month < 0) {  y = CalendarUtils.*floorDivide*(month, 12);  month = CalendarUtils.*mod*(month, 12);  }  BaseCalendar.Date d = getCalendarDate();  **if** (y != 0) {  d.setNormalizedYear(d.getNormalizedYear() + y);  }  d.setMonth(month + 1); // adjust 0-based to 1-based month numbering  }  @Deprecated  **public** **int** ~~getDate~~() {  **return** normalize().getDayOfMonth();  }  @Deprecated  **public** **void** ~~setDate~~(**int** date) {  getCalendarDate().setDayOfMonth(date);  } |

对实例域（成员变量）作出修改的方法称为更改器方法（mutator method），仅仅访问实例域（成员变量）而不进行修改的方法称为访问器方法（accessor method）。通常更改器方法以get开头，我们也称为getter方法，访问器方法以set开头，我们也称为setter方法。

注意：如果我们的实例域为boolean类型，则getter方法以is开头。

### 使用自定义类

先前我们使用了基础类库中的String类，也使用了基础类库中的Date类。很多情况下我们需要根据我们的需求定义自己的类。也就是自定义类，我们来看下自定义类的语法：

[public|protected|private] class ClassName [extends ParentClassName] [implements Interface1[,Interface2[,Interface3]]]{

field1;

field2;

…

constructor1;

constructor2;

…

method1;

method2;

…

}

下面是一个自定义的Person的类：

|  |
| --- |
| **import** java.util.Date;  **public** **class** Person {  **private** **long** id;  **private** String name;  **private** **int** age;  **private** **boolean** checked;  **private** Date birthday;  **private** **double** salary;    **public** Person() {  **super**();  }  **public** Person(String name) {  **super**();  **this**.name = name;  }  **public** Person(String name, **int** age, **boolean** checked, Date birthday, **double** salary) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.checked = checked;  **this**.birthday = birthday;  **this**.salary = salary;  }  **public** **long** getId() {  **return** id;  }  **public** **void** setId(**long** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  **public** **boolean** isChecked() {  **return** checked;  }  **public** **void** setChecked(**boolean** checked) {  **this**.checked = checked;  }  **public** Date getBirthday() {  **return** birthday;  }  **public** **void** setBirthday(Date birthday) {  **this**.birthday = birthday;  }  **public** **double** getSalary() {  **return** salary;  }  **public** **void** setSalary(**double** salary) {  **this**.salary = salary;  }      } |

#### 构造器

我们看下，我们自定义的Person类中有三个构造器：

|  |
| --- |
| **public** Person() {  **super**();  }  **public** Person(String name) {  **super**();  **this**.name = name;  }  **public** Person(String name, **int** age, **boolean** checked, Date birthday, **double** salary) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.checked = checked;  **this**.birthday = birthday;  **this**.salary = salary;  } |

构造器或者说构造方法必须与类的名称相同，没有返回值。在构造对象的时候构造器会先运行，将实例域初始化为我们希望的状态。构造器总是伴随着new的操作而执行，我们无法对一个已经存在的对象调用构造器方法：

xiaoming.Person()；//错误

构造器的特点如下：

* 1. 构造器与类同名
  2. 每个类可以有一个以上的构造器
  3. 构造器可以有0个，1个或者多个参数。
  4. 构造器没有返回值
  5. 构造器总是伴随着new操作一起调用

##### 重载

我们看到我们自定义的Person类有3个构造器，3个构造器的参数格式不同。这在面向对象里面叫做重载（overload）。如果多个方法具有相同的名称，不同的参数就是重载。编译器必须在编译的时候挑选出选择哪个方法。

注意：

返回值不是方法签名的一部分，也就是说，不能有两个名称相同、参数类型也相同但返回值不同的方法。

##### 默认初始化

如果在构造器中没有显示的给域赋予初值，那么就会被自动的赋为默认值：

数值为0，布尔为false，对象为null。

注意：

只有实例域会被赋予默认的初始值，局部变量不会。

##### 无参构造器

很多类都包含一个无参的构造器。对象由无参构造器创建，实例域会被赋予适当的默认值。如果自定义类的时候没有编写构造器，那么编译器会默认的给添加一个无参的构造器。该构造器将所有实例域设置为默认值。

如果类提供了至少一个构造器，但是没有无参构造器，那么再使用无参构造器构造对象是不合法的。

注意：

当且仅当类没有提供任何构造器的时候，系统才会提供一个默认的构造器。

##### 显示初始化

由于类的构造器方法可以重载，所有可以采用多种方式设置实例域的初始状态。确保不管怎样调用构造器，每个实例域都可以被设置为一个有意义的初值。如：

class Person{

private String name = “”;

}

##### 参数名

|  |
| --- |
| **public** Person(String name, **int** age, **boolean** checked, Date birthday, **double** salary) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.checked = checked;  **this**.birthday = birthday;  **this**.salary = salary;  } |

注意区分我们的实例域变量名称和方法的参数名称。this关键字用来指示当前对象。

##### 调用另外构造器

关键字this除了可以做为隐式参数表示当前对象还是调用其他的构造器：

|  |
| --- |
| **public** Person(String name, **int** age, **boolean** checked, Date birthday, **double** salary) {  **this**(name);  **this**.age = age;  **this**.checked = checked;  **this**.birthday = birthday;  **this**.salary = salary;  } |

关键字super可以调用父类的构造器：

|  |
| --- |
| **public** Person(String name) {  **super**();  **this**.name = name;  } |

注意：

如果一个类没有显示的继承父类，则默认继承Object类。

##### 再说初始化块

初始化实例域有三种方式：

* 构造器中初始化
* 声明的时候初始化
* 通过初始化块初始化

接下来来说通过初始化块初始化。

初始化块（initialization block）：用大括号包含起来的代码块。只要构造器执行就会调用该代码块：

|  |
| --- |
| **public** **class** Test {    **public** **static** **void** main(String[] args) {  Person person = **new** Person("张三",12);  System.***out***.println(person.getName());  }  }  **class** Person {    **private** String name;    **private** **int** age;    {  System.***out***.println("初始化块代码开始执行....................");  }  **public** Person(String name, **int** age) {  **super**();  System.***out***.println("构造方法开始执行....................");  **this**.name = name;  **this**.age = age;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  }  控制台打印结果如下：  初始化块代码开始执行....................  构造方法开始执行....................  张三 |

注：静态初始化块用于初始化静态变量。这里不予详说。

调用构造器的处理步骤：

1. 所有数据域被初始化为默认值（0，false，null）
2. 按照在类声明中的出现次序，执行所有在类中出现的初始化语句和初始化块
3. 如果构造器的第一行调用的其他的构造器，则执行其他被调用的构造器
4. 执行构造器的初始化主体

#### 隐式参数和显示参数

在java中，每个非静态方法都会由编译器默认传入一个隐式的参数this表示当前的对象。其他的在参数列表中声明的参数是显示参数。

#### 私有方法

私有方法以private关键字修饰，表示只有该类内部可以访问。

#### final实例域

实例域可以声明为final类型，表示对象构建后该实例域就不能被修改。

#### 静态域静态方法

**静态域：**如果将类中的域定义为static，则这个域属于这个类，而不属于这个类的某个对象，每个类中只有 一个这样的域，而每一个类对象对于所有的实例域(即没有定义为static的域)都有自己的一份拷贝。例如：

class Employee

{

　private int id;

　private static int nextId = 1;

}

如果有1000个Employee对象，则有1000个实例域id，但是只有一个静态域nextId；即使没有一个Employee对象，静态域nextId也存在，它属于类，不属于任何对象。

静态域也称为类域。

**静态常量**：如果一个域被定义为static final，则这个域就是一个静态常量。不能省略任何一个关键字，若是少了static，则该域变成了一个实例域，需要由类对象对其进行访问。若是省略了final，则该域变成了静态域，静态方法可以对其进行修改。

例子：

public class Math

{

　　public static final double PI=3.14159265358979323864；

}

其中PI就是一个静态常量。

**静态方法**：静态方法是一种不能向对象实施操作的方法。Math的pow方法就是一个静态方法，在运算时，不使用任何Math对象，换句话说，没有隐式的参数this。因为静态方法不能操作对象，所以不能在静态方法中访问实例域，但是静态方法可以访问自身类中的静态域。可以使用对象调用静态方法，但是这样容易引起混淆，因为计算的结果与对象毫无关系，建议还是使用类名，而不是类对象调用静态方法。

例如：

public static int getNextId( )

{

　　return nextId；

}

但是，如果去掉其中的关键字static，它就成了非静态方法，但是也可以访问类中的静态域，这时就需要由该类的对象来调用该函数。

在下面两种情况下使用静态方法：

1.一个方法不需要访问对象的状态，其所需的参数都是通过显式的提供

2.一个方法只需访问类的静态域

### 说说this关键字

this关键字总是指向调用该方法的对象。根据this出现位置的不同，this作为对象的默认引用有两种情况：

·构造器中引用该构造器正在初始化的对象；

·在方法中引用调用该方法的对象。

this可以代表任何对象，当this出现在某个方法体中时，它所代表的对象是不确定的，但它的类型是确定的，它所代表的对象只能是当前类；只有当这个对象被调用时，它所代表的对象才被确定下来：谁在调用这个方法，this就代表谁。

大部分的时候，一个方法访问该类中定义的其他方法、Field时加不加this前缀的效果是完全一样的。

对于static修饰的方法而言，则可以使用类来直接调用该方法，如果在static修饰的方法中使用this关键字，则这个关键字就无法指向任何对象。所以，static修饰的方法中不能使用this引用。由于static修饰的方法不能使用this引用，所以static修饰的方法不能访问不使用static修饰的普通成员，因此Java语法规定：静态成员不能直接访问非静态成员。

如果确实需要在静态方法中访问另一个普通方法，则只能重新创建一个对象。

大部分的时候，普通方法访问其他方法、Field时无须使用this前缀，但如果方法里有个局部变量和Field同名，但程序又需要在该方法里访问这个被覆盖的Field，则必须使用this前缀。

### 参数传递

在程序设计语言中，参数的传递包含按值调用和按引用调用两种方式。按值调用表示方法接收的是方法调用者提供的值。按引用调用表示方法接收到的是调用者提供的变量地址。一个方法可以修改按引用调用传递进来变量的值，而不能修改按值调用传递进来变量的值。

在java语言中方法调用只有按值调用一种方式。也就是说方法得到的是参数值的拷贝。方法不能修改传递给他的任何参数变量的内容。是基本类型，就拷贝一个基本类型传进方法；是引用，就拷贝一个引用变量传进方法;

注意：

对于8种基本数据类型、String对象，采用复制一份传递到方法中的方式，方法中无法修改传递的内容本身。

对于引用类型（类类型，数组类型），java传递的是地址，可以修改地址引用位置的数据内容，但不能修改地址本身。

### 包

包（package）是Java提供的一种区别类的名字空间的机制，是类的组织方式，是一组相关类和接口的集合，它提供了访问权限和命名的管理机制。

Java中提供的包主要有以下3种用途:

1) 将功能相近的类放在同一个包中，可以方便查找与使用。

2) 由于在不同包中可以存在同名类，所以使用包在一定程度上可以避免命名冲突。

3) 在Java中，某次访问权限是以包为单位的。

创建包可以通过在类或接口的源文件中使用 package语句实现，package语句的语法格式如下：

package 包名;

包名：必选，用于指定包的名称，包的名称为合法的 Java标识符。包名一般采用公司域名的反转。比如我有一个abc.com的域名，那么报名可以采用：com.abc的方式。

在类有类包约束之后， 类可以访问其所在包中的所有类，还可以使用其他包中的所有public类。访问其他包中的public类可以有以下两种方法。

**1) 使用长名引用包中的类**

使用长名引用包中的类比较简单，只需要在每个类名前面加上完整的包名即可。例如：

java.util.Date myBirthday = new java.util.Date();

**2) 使用import语句引入包中的类**

由于采用使用长名引用包中的类的方法比较繁琐，所以Java提供了import语句来引入包中的类。import语句的基本语法格式如下：

import 包名1[.包名2.……].类名|\*;

如：

import java.util.Date;

import java.util.\*;

注：

静态导入，import static java.lang.System.\*;这种方式之后就可以使用System的静态方法和静态域，而不需要System前缀。如：out.println(“abc”);