# Java语法

# 第一篇 Java语言基础

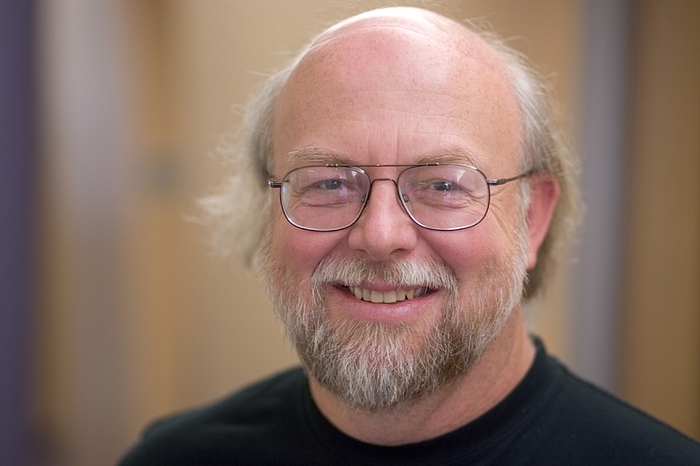
## 第一章 Java语言入门

### 第一节 Java语言概述

#### 1. Java语言简介

**Java语言**是美国Sun公司（Stanford University Network），在1995年推出的高级的编程语言。所谓编程语言，是计算机的语言，人们可以使用编程语言对计算机下达命令，让计算机完成人们需要的功能。

下图是java语言之父 **James Gosling（詹姆斯·高斯林）**



#### 2. Java语言的发展史

* **1995 年Sun公司发布Java1.0版本**
* 1997 年发布Java 1.1版本
* 1998 年发布Java 1.2版本
* 2000 年发布Java 1.3版本
* 2002 年发布Java 1.4版本
* 2004 年发布Java 1.5版本
* 2006 年发布Java 1.6版本
* **2009 年Oracle甲骨文公司收购Sun公司，并于2011发布Java 1.7版本**
* 2014 年发布Java 1.8版本
* **2017 年发布Java 9.0版本(本文档版本)**

#### 3. Java语言能做什么

Java语言主要应用在互联网程序的开发领域。常见的互联网程序比如天猫、京东、物流系统、网银系统等，以及服务器后台处理大数据的存储、查询、数据挖掘等也有很多应用。

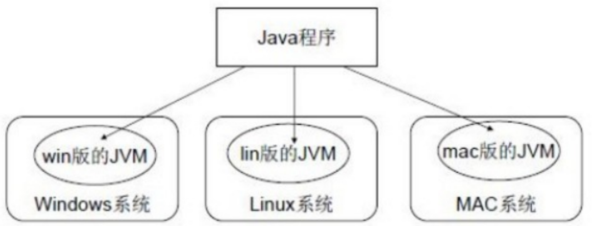
### 第二节 Java原生开发环境

#### 1. Java虚拟机(JVM)

**JVM**（Java Virtual Machine ）：Java虚拟机，简称JVM，是运行所有Java程序的假想计算机，是Java程序的运行环境，是Java 最具吸引力的特性之一。我们编写的Java代码，都运行在 JVM 之上。

#### 2. Java的跨平台特性

任何软件的运行，都必须要运行在操作系统之上，而我们用Java编写的软件可以运行在任何的操作系统上，这个特性称为Java语言的跨平台特性。该特性是由JVM实现的，我们编写的程序运行在JVM上，而JVM运行在操作系统上。



如图所示，Java的虚拟机本身不具备跨平台功能的，每个操作系统下都有不同版本的虚拟机。

#### 3. JRE和JDK

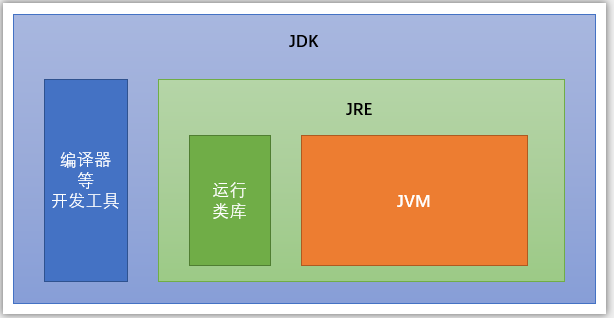
* **JRE** (Java Runtime Environment)

Java程序的运行时环境，包含 JVM 和运行时所需要的 核心类库 。

* **JDK** (Java Development Kit)

Java程序开发工具包，包含 JRE 和开发人员使用的工具。

JRE和JDK的关系如下图所示：



总结：我们想要运行一个已有的Java程序，那么只需安装 JRE 即可。我们想要开发一个全新的Java程序，那么必须安装 JDK 。

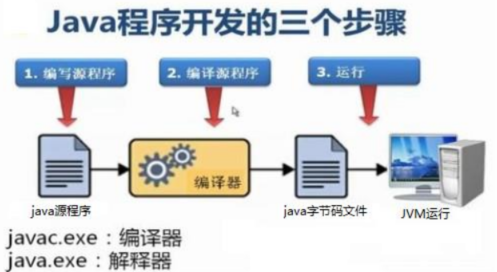
注意：JDK安装后还需配置环境变量

**JAVA\_HOME** = JDK安装路径(如D:\java\jdk-9.0.1)

**PATH** += %JAVA\_HOME%\bin

#### 4. 第一个Java程序

Java程序开发三步骤为：**编写 -> 编译 -> 运行**。



**(1) 编写源文件**

在一个方便的目录新建后缀名为.java的HellWorld文件，输入以下代码：

public class **HelloWorld** {

       public static void **main**(String[] args) {

         System.out.**println**("Hello World!");

       }

    }

注意：文件名必须是 HelloWorld ，保证文件名和类的名字是一致的，注意大小写。

附HelloWorld.java文件如下：



**(2) 编译Java源文件**

在DOS命令行中，**进入Java源文件的目录**，使用 javac 命令进行编译。命令格式如下：

javac Java源文件名.后缀名 例：javac HelloWorld.java

编译成功后，命令行没有任何提示。打开文件目录，发现产生了一个新的文件 HelloWorld.class ，该文件就是编译后的文件，是Java的可运行文件，称为**字节码文件**，有了字节码文件，就可以运行程序了。

注意：Java源文件的编译工具 javac.exe ，在JDK安装目录的bin目录下。但是由于配置了环境变量，可以再任意目录下使用。

**(3) 运行java程序(字节码)**

在DOS命令行中，进入Java源文件的目录，使用 java 命令进行运行。命令格式如下：

java 类名字 例：java HelloWorld

注意：java HelloWord **不要写 不要写 不要写** .class

**(4) HelloWorld的几点说明**

* **main方法**

**main方法**又称为主方法。写法是**固定格式**不可以更改。main方法是程序的入口点或起始点，无论我们编写多少程序，JVM在运行的时候，都会从main方法这里开始执行。

* **java的注释**

**注释**就是对代码的解释和说明。其目的是让人们能够更加轻松地了解代码。为代码添加注释，是十分必须要的，它不影响程序的编译和运行。Java 中有单行注释和多行注释：

* 单行注释：以 **//** 开头 **换行** 结束
* 多行注释：以 **/\*** 开头以 **\*/** 结束
* **Java的关键字(KeyWords)**

**关键字**是指在程序中，Java已经定义好的单词，具有特殊含义。HelloWorld 案例中，出现的关键字有 *public 、 class 、 static 、 void* 等，这些单词已经被Java定义好，**全部都是小写字母**，notepad++中颜色特殊。

* **Java的标识符**

**标识符** 是指在程序中，我们自己定义内容。比如类的名字、方法的名字和变量的名字等等，都是标识符。HelloWorld 案例中，出现的标识符有类名字 *HelloWorld* 。标识符必须符合命名规则，如下：

* 标识符只可以包含 **英文字母(区分大小写)**、**数字** 、**$（美元符号）**和**\_（下划线）**。
* 标识符不能以数字开头且不能是关键字。

对于标识符的命名除了以上硬性要求外，还有些软性建议：

① 类名和方法名分别采用大小驼峰式命名规范，变量名全部字母小写。

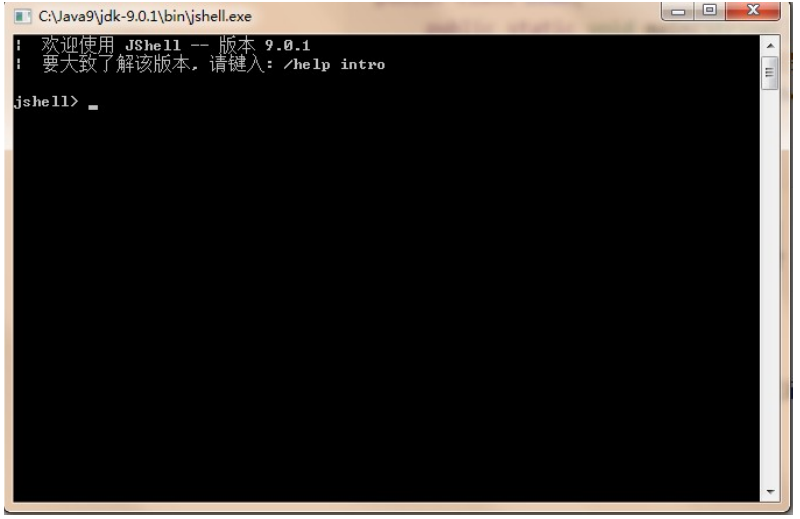
② 一般不在标识符中使用$美元符号，因为它在内部类处有特殊含义。

#### 5. Jshell脚本

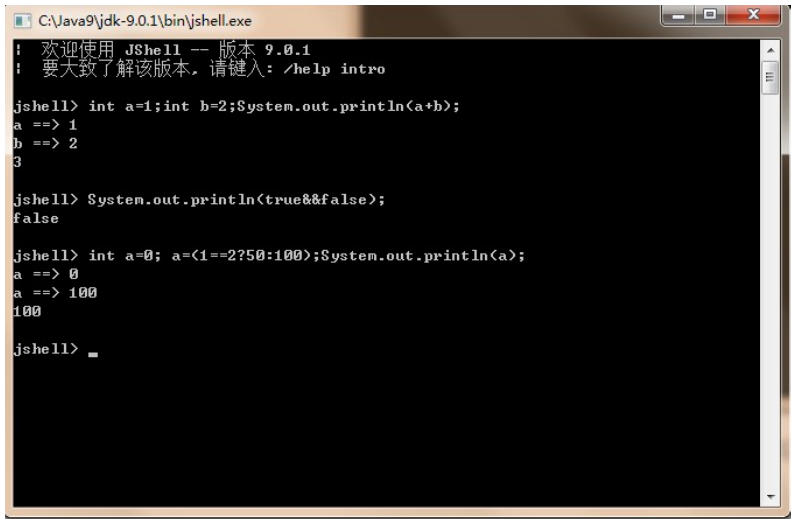
什么时候会用到 JShell 工具呢，当我们编写的代码非常少的时候，而又不愿意编写类，main方法，也不愿意去编译和运行，这个时候可以使用JShell工具。

* **如何使用Jshell**

(1) 启动JShell工具，在DOS命令行直接输入JShell命令。



(2) 接下来可以编写Java代码，无需写类和方法，直接写方法中的代码即可，同时无需编译和运行，直接回车即可。

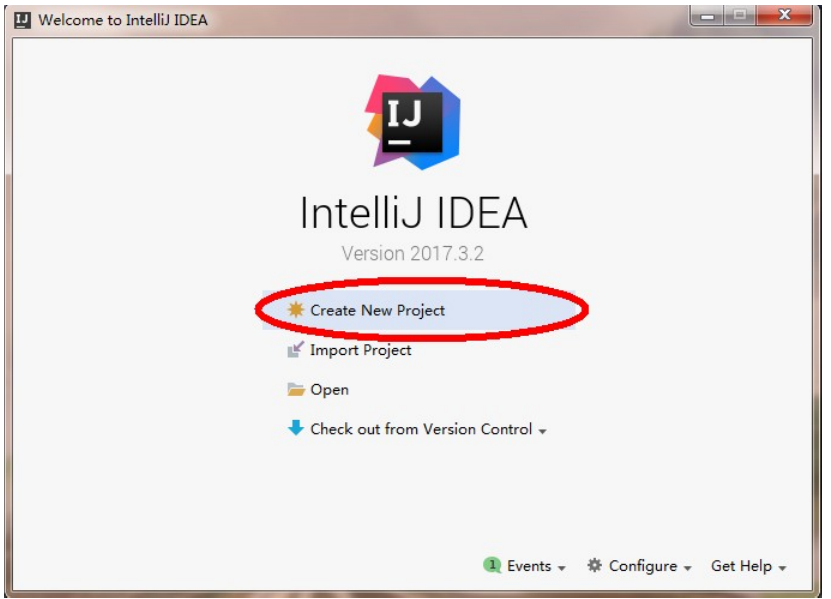


### 第三节 Java集成开发环境

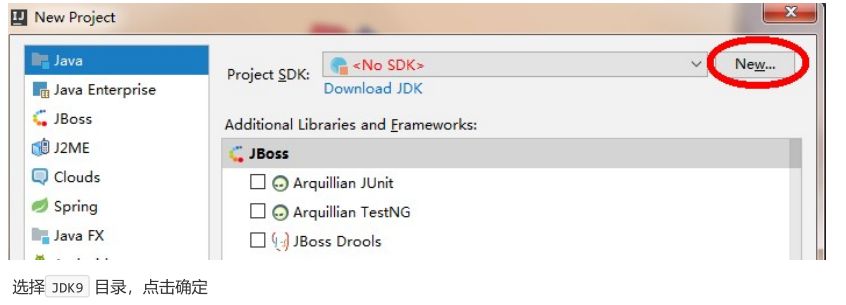
Java集成开发环境有很多，其中使用最广泛的就是IntelliJ IDEA。IDEA是一个专门针对Java的集成开发工具(IDE)，由Java语言编写。

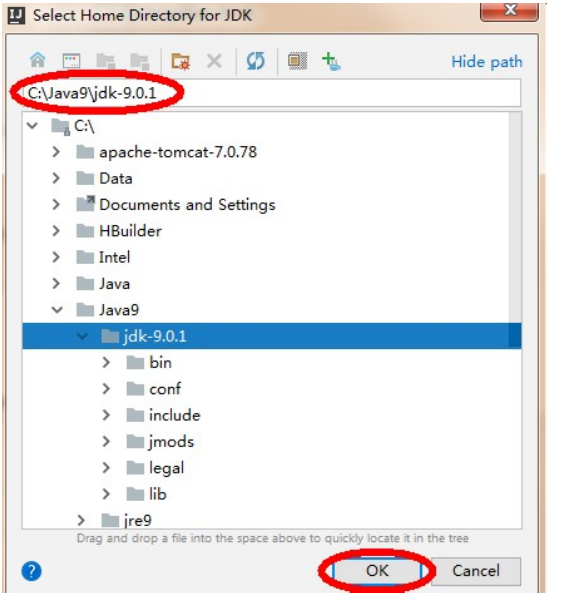
#### 1. 使用IDEA运行HelloWorld程序

(1) 打开IDEA ，选择 **Create New Project**

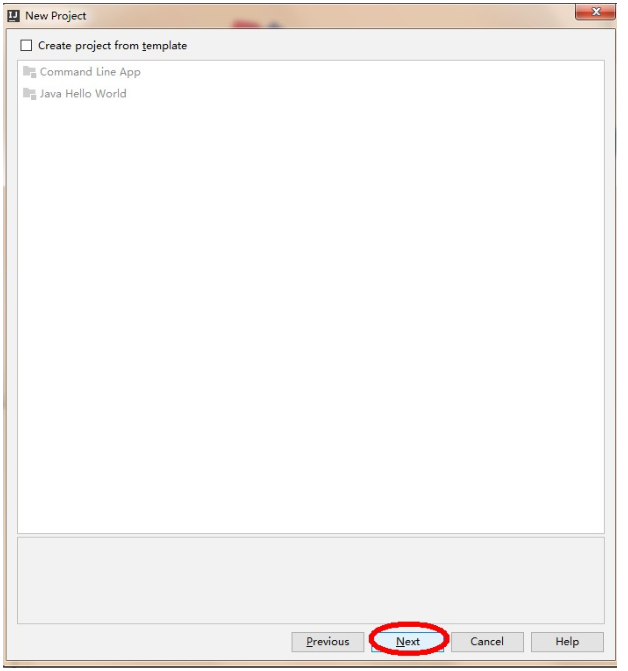
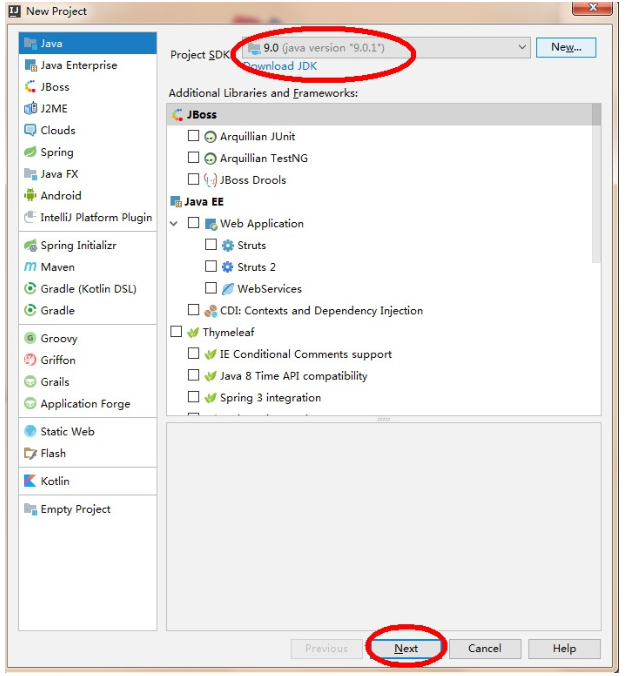


(2) 点击 **new** 按钮，配置安装的 JDK9 版本

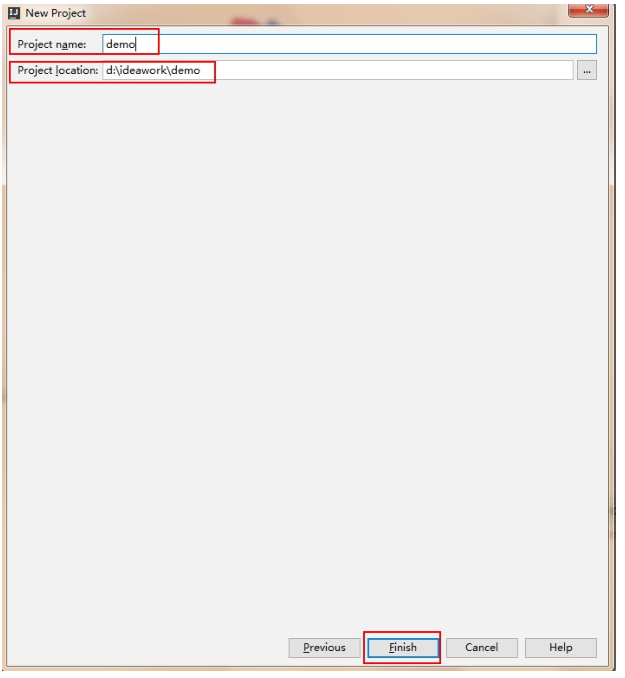




(3) 不使用任何模板，点击next -> next

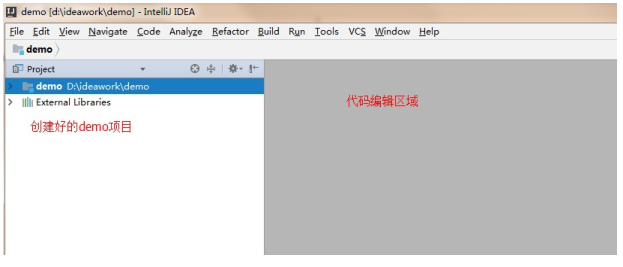


(4) 指定工程名和目录(如果没有该目录，会自动创建)



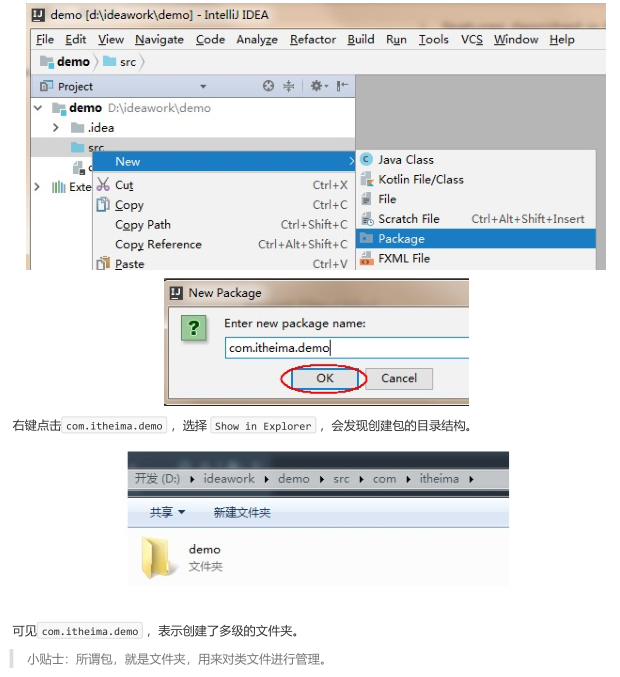
注意：首次新建项目时，默认的Project Location路径可能会有问题，如 c:\\xxx ，正确写法为 c:\xxx，更改一次后不会再出现此类问题。

到此为止，我们的项目已经创建好了，可以看到IDEA的工作界面，如果再新建项目，点击 File ->new->Project。



**(5) 创建一个Java包**

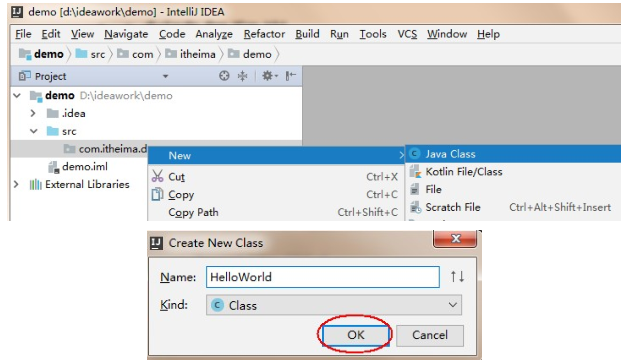
展开创建的工程/模块，在**源代码目录 src** 上，鼠标右键，选择 **new ->package** ，键入包名com.itheima.demo，点击确定。



**所谓包，就是文件夹，用来对类文件进行管理，**右击点击com.ittheima.demo,选择Show in Explorer，可以看到创建包的目录结构

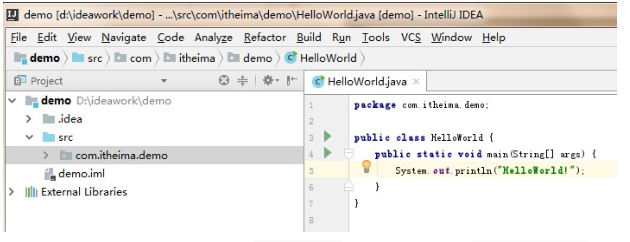
**(6) 创建一个HelloWorld类**

在创建好的包上，鼠标右键，选择  **new ->class** 创建类，键入类名.



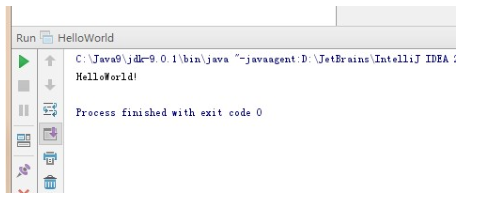
**(7) 代码编辑**

在代码编辑区，键入主方法，并输出 HelloWorld 。



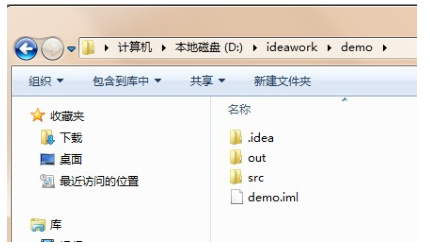
**(8) 运行程序**

在代码编辑区鼠标右键，选择 Run HelloWorld 即可，或在菜单中选择 Run ->Run HelloWorld 。可以看到输出结果如下：

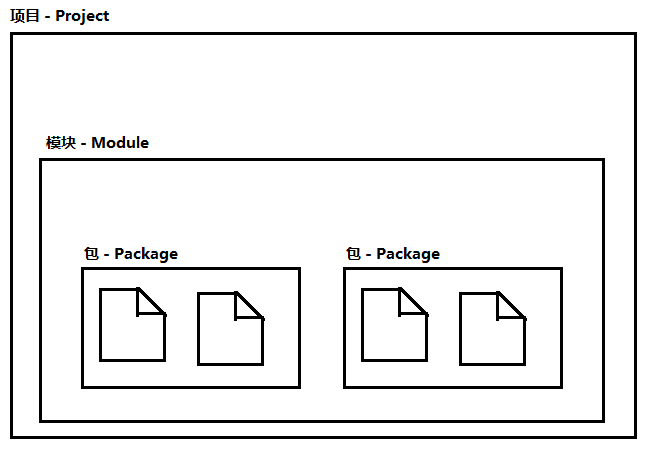


#### 2. IDEA项目结构

我们创建的项目，保存在 d:\ideawork目录的demo下，其中.idea目录和demo.iml文件和我们开发无关，是IDEA工具自己使用的，out 目录是存储编译后的.class文件，src 目录就是存储我们编写的.java源文件的地方，打开后可以看到我们建好的com.itheima.demo包。



以上是IDEA项目在我们计算机上的目录结构，实际上，Java程序基本上都遵循以下的代码组织方式，



最外层为Project，然后有许多大的Module，而每个Module都由一个个Package组成，我们导入源代码一般以包为单位，叫做导包，而使用源代码则是调用包中的class文件，即java类。

#### 3. IDEA常用快捷键

IDEA的快捷键非常之多，我们记住几个常用的即可：

**(1) 代码生成类快捷键**

* **Alt+Insert**

自动生成代码框架，如get、set方法和构造器等。

* **Alt+Enter**

智能修正报红的代码，如果自动导入包等。也可以用于在实现抽象方法时直接生成方法框架。

* **次数.fori/次数.forr**

键入以上代码后回车，IDEA会自动帮我们生成一个循环指定次数的for循环/反向for循环。

* 容器名.fori/容器名.forr

键入以上代码后回车，IDEA会自动帮我们生成该容器的遍历代码/反序遍历代码。

(2) 代码编辑类

* Ctrl+C/Ctrl+V/Ctrl+Y/Ctrl+D

分别为复制、粘贴、复制光标所在行、复制光标所在行并插入光标的下一行

* Ctrl+/以及Ctrl+Shift+/

分别为单行注释和多行注释，重复操作极为取消注释

Ctrl+Shift+上下方向键

上下移动光标所在行代码。

* Ctrl+Alt+L

格式化选中代码

* Shift+F6

在文件视图中选中一个类，按此键可以给方法改名。

提示：文件视图中的类可以直接复制的，稍作修改即可。

* **Alt+/(原Ctrl+Space)**

用来手动打开代码提示，但这个快捷键和Windows中的输入法切换快捷键冲突，需要修改IDEA中的快捷键，修改位置在 *File ->Settings->keymap->Main menu->code->Completion->Basic***。**

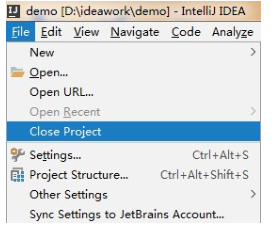
**(3) 其它类**

* Ctrl+左键

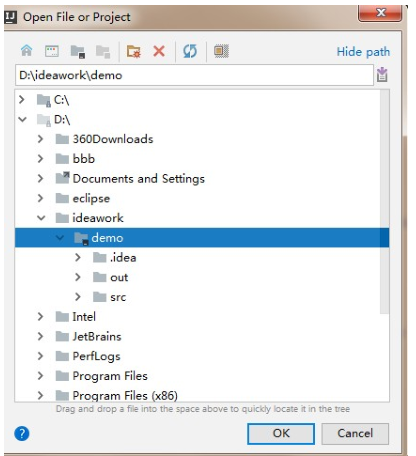
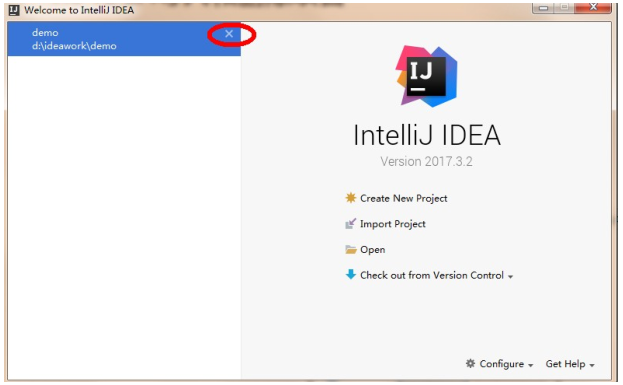
转到该对象的定义处。

#### 4. IDEA项目的关闭和打开

**(1) 关闭IDEA中已经存在的项目**:File ->Close Project



**(2) 打开一个已有的IDEA项目：**在IDEA的启动界面上，点击 OPEN ，选择项目目录即可



Tips1：右键src不能新建pakeage的解决方案：

右击src -> Mark Directory As –> Sources Root

Tips2：若想通过 IDEA同时开启多个项目，点击OPEN打开项目时，点击New Window按钮

## 第二章 Java数据类型

### 第一节 java中的六类常量

**常量**是指在Java程序中固定不变的数据。Java中的常量有六种，分别是：

* **整数常量**

包含所有的整数，如*0，567，-9，100L*。

* **小数常量**

包含所有的小数，如*0.0，-0.1，2.55F*。

* **字符常量**

单引号引起来的常量，单引号内**有且只能**写一个字符，如’a’,’ ‘,’汉’。

tips1：双引号内不写或写多个字符都是错误的字符常量，如~~’’~~，~~’ab’~~等都是错误的。

tips2：在java中，字符采用unicode编码，占两个字节，并且一个汉字只算一个字符。

* **字符串常量**

双引号引起来的常量，可以写多个字符，也可以什么都不写，代表空串，如”Hello”，””。

* **布尔常量**

只有两个值的常量，分别为true和flase。

Tips1：特别注意，在java中，布尔常量不能用0和非0来表示。

* **空常量**

空常量代表(null)，表示未知，不能用来打印输出，如·~~System.out.println(null)~~是错误的。

### 第二节 变量与数据类型

**变量**是在程序中可以变化的量。Java中要求一个变量每次只能保存一个数据，并且必须要明确保存的数据类型。Java中数据类型分为基本数据类型和引用数据类型。

#### 1. 基本数据类型

Java中的基本数据类型可分为**四类八种**，分别是：

* **整型变量**

整型变量从小到大又可区分为**字节型(byte)、短整型(short)、整型(int)和长整型(long)**四种，分别占用1、2、4和8个字节，java中默认的整型变量是int类型。

* **浮点型变量**

浮点型变量有单精度浮点型(float)和双精度浮点型(double)两种，分别占用4和8个字节，默认的浮点型变量是double类型。

tips：特别需要注意的一点，数据表示范围与字节数不一定正相关，例如float虽然只占用了4个字节，但数据表示范围却比占用8个字节的long广泛的多。

* **字符型变量**

字符型变量只有char一种，其采用unicode方式编码，***占用两个字节***，取值范围为0-65535。

* **布尔型变量**

布尔型变量(boolean)只有true和flase两种值，占用一个字节，用来表示真或假。

#### 2. 引用数据类型

引用数据类型包含除基本数据类型中的四类八种外所有的类型，如类、数组、接口、**字符串**、lamda表达式等。

#### 3. 变量的定义

Java中的变量必须先定义后才能使用，定义的格式如下：

数据类型 变量名 = 数据值;

下面是所有基本数据类型变量的定义示例：

public class **Variable** {

    public static void **main**(String[] args){

*//定义字节型变量*

            byte b = 100;

            System.out.**println**(b);

*//定义短整型变量*

            short s = 1000;

            System.out.**println**(s);

*//定义整型变量*

            int i = 123456;

            System.out.**println**(i);

*//定义长整型变量*

            long l = 12345678900L;

            System.out.**println**(l);

*//定义单精度浮点型变量*

            float f = 5.5F;

            System.out.**println**(f);

*//定义双精度浮点型变量*

            double d = 8.5;

            System.out.**println**(d);

*//定义布尔型变量*

            boolean bool = false;

            System.out.**println**(bool);

*//定义字符型变量*

            char c = 'A';

            System.out.**println**(c);

    }

}

tips1：在java中，**未初始化的变量无法使用**，但给变量初始化时不要超出其数据类型所能表示的范围，否则编译器将会报错，如错误的定义：~~byte num4 = 400;~~

tips2：可以通过一个语句来创建多个变量，但是一般情况不推荐这么写,如int a=1,b=2;

### 第三节 数据类型转换

Java程序中要求参与的计算的数据，**必须要保证数据类型的一致性**，如果数据类型不一致将发生类型的转换，转换可分为自动类型转换和强制类型转换。

#### 1. 自动类型转换

**自动转换是**将 取值范围小的类型 自动提升为 取值范围大的类型 。如以下示例：

public static void **main**(String[] args) {

        int i = 1; byte b = 2;

*// byte x = b + i; // int类型和byte类型运算，结果是int类型，因此报错*

        int j = b + i;

         System.out.**println**(num1); *// 100*

*// 左边是double类型，右边是float类型，左右不一样*

        double num2 = 2.5F; *// float --> double，符合从小到大的规则，发生了自动类型转换*

        System.out.**println**(num2); *// 2.5*

*// 左边是float类型，右边是long类型，左右不一样*

        float num3 = 30L; *// long --> float，范围是float更大一些，符合从小到大的规则*

        System.out.**println**(num3); *// 30.0*

}

byte 类型内存占有1个字节，在和 int 类型运算时会提升为 int 类型 ，自动补充3个字节，因此计算后的结果还是 int 类型。long和float参与运算也是同样的道理。

Tips1：特别的是，对于byte 、short、char 来说，只要参与运算，至少会被提升为int类型。

tips2: boolean类型不能发生数据类型转换.

#### 2. 强制转换转换

先来看看将 1.5 赋值到 int 类型变量会发生什么？

int i = 1.5; *// 错误*

因为java是强类型检查语言，并且上例不符合取值范围从小到大的自动转换原则，结果显然为编译失败，想要赋值成功，必须通过强制类型转换，将 double 类型强制转换成 int 类型。

**强制类型转换是**将 取值范围大的类型 强制转换成 取值范围小的类型 。比较而言，自动转换是Java自动执行的，而强制转换需要我们自己手动执行。强制类型转换的格式如下：

数据类型 变量名 = （数据类型）被强转数据值；

因此将 1.5 赋值到 int 类型，代码需要修改为：

int i = (int)1.5; *// double类型数据强制转成int类型，直接去掉小数点。*

同样道理，当一个 short 类型与 1 相加，我们知道会类型提升，但是还想给结果赋值给short类型变量，就需要强制转换。

public static void **main**(String[] args) {

         short s = 1;

         s = s + 1；*//编译失败，因为s和1做运算的时候，1是int类型，s会被提升为int类型*

         s = (short)(s+1);*//编译成功*

    }

**强制类型转换是有风险的**

① 强制类型转换可能会造成精度损失。

② 强制类型转换可能会造成数据溢出。

如下例子：

*// double --> int*

int num3 = (int) 3.99;

System.out.**println**(num3); *// 3 舍弃了小数部分，发生了精度损失。*

*// long --> int*

int num2 = (int) 6000000000L;

System.out.**println**(num2); *// 1705032704 发生了数据溢出*

#### 3. 特别说明1：编译器隐含的强制类型转换

对于byte/short/char三种类型来说，如果右侧赋值的数值没有超过范围，那么javac编译器将会自动隐含地为我们补上一个(byte)(short)(char)。但如果右侧超过了左侧范围，那么直接编译器报错。

public class **Demo12Notice** {

    public static void **main**(String[] args) {

*// 右侧确实是一个int数字，但是没有超过左侧的范围，就是正确的。*

*// int --> byte，不是自动类型转换*

        byte num1 = */\*(byte)\*/* 30; *// 右侧没有超过左侧的范围*

        System.out.**println**(num1); *// 30*

*// byte num2 = 128; // err：右侧超过了左侧的范围*

*// int --> char，没有超过范围*

*// 编译器将会自动补上一个隐含的(char)*

        char zifu = */\*(char)\*/* 65;

        System.out.**println**(zifu); *// A*

    }

}

#### 4. 特别说明2：编译器的常量优化

下面的程序有问题吗？

public static void **main**(String[] args){

      byte b1=1; byte b2=2;

      byte b3=1 + 2;//OK

      byte b4=b1 + b2;//err

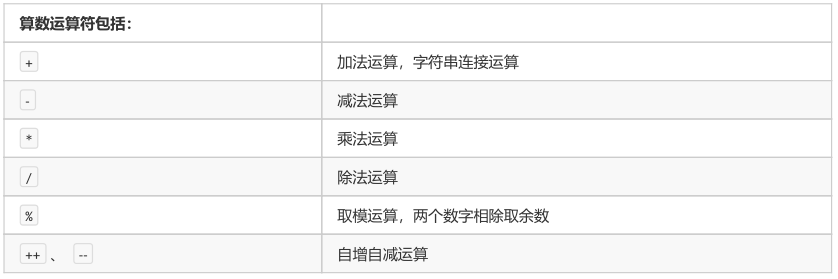
    }

**分析：** b3 = 1 + 2 ， 1 和 2 是常量，为固定不变的数据，在编译的时候（编译器javac），已经确定了 1+2 的结果并没有超过byte类型的取值范围，因此会补上(byte)赋值给变量 b3 ，因此 b3=1 + 2 是正确的。

反之， b4 = b2 + b3 ， b2 和 b3 是变量，变量的值是可能变化的，在编译的时候，编译器javac不确定b2+b3的结果是什么，因此会直接将结果以int类型进行处理，所以int类型不能赋值给byte类型，因此编译失败。

## 第三章 运算符

### 第一节 算术运算符



tips1：只有整数参与的算术运算，不可能出现小数。

tips2：对负数进行取模运算，结果的正负始终与前一个数值相同。

**四则运算当中的加号“+”有常见的三种用法：**

(1) 对于数值来说，那就是加法。

(2) 对于字符char类型来说，在计算之前，char会被提升成为int，然后再计算。

(3) 对于字符串String（首字母大写，并不是关键字）来说，加号代表字符串连接操作。

public class **Demo05Plus** {

    public static void **main**(String[] args) {

        String str2 = "Java";

        System.out.**println**(str2 + 20); *// Java20  // String + int --> String*

*// 注意优先级问题*

        System.out.**println**(str2 + 20 + 30); *// Java2030  // String + int + int*

        System.out.**println**(str2 + (20 + 30)); *// Java50  // String      + int*

    }

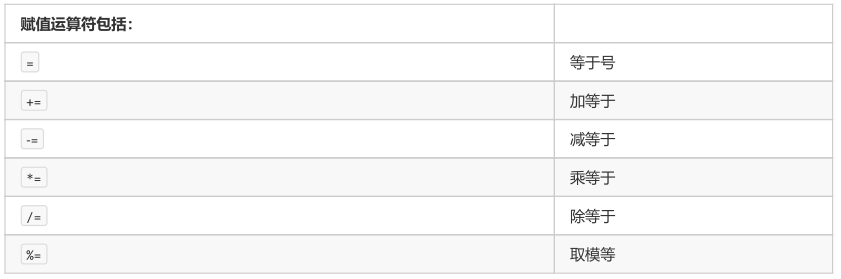
}

**自增自减运算符的使用说明**

自增自减运算符可以单独使用，也可以混合使用。在单独使用的时候，前++和后++没有任何区别。在混合使用时，*【前++】*表示*【先加后用】*，*【后++】*表示*【先用后加】。*

Tips：只有变量才能使用自增、自减运算符。常量不可发生改变，所以不能用。

### 第二节 赋值运算符



赋值运算符，就是将符号右边的值，赋给左边的变量。

public static void **main**(String[] args){

        int i = 5;

        i+=5;*//计算方式 i=i+5 变量i先加5，再赋值变量i*

        System.out.**println**(i); *//输出结果是10*

    }

**复合赋值运算符其中隐含了一个强制类型转换**

下面的程序有问题吗？

public static void **main**(String[] args){

      short s = 1;

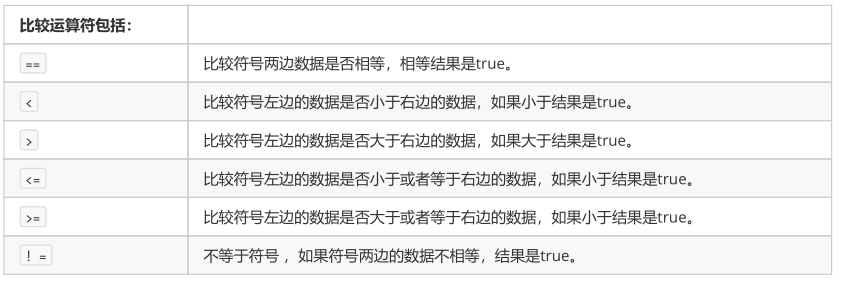
      s+=1;

      System.out.**println**(s);

    }

**分析：**s += 1 逻辑上看作是 s = s + 1 计算结果被提升为int类型，再向short类型赋值时发生错误，因为不能将取值范围大的类型赋值到取值范围小的类型。但是，s=s+1 进行两次运算 ， += 是一个运算符，只运算一次，并带有强制转换的特点，**也就是说 s += 1 就是 s = (short)(s + 1)** ，因此程序没有问题编译通过，运行结果是2.

### 第三节 比较运算符



比较运算符，是两个数据之间进行比较的运算，运算结果都是布尔值 true 或者 false 。

public static void **main**(String[] args) {

        System.out.**println**(1==1);*//true*

        System.out.**println**(1<2);*//true*

        System.out.**println**(3>4);*//false*

        System.out.**println**(3<=4);*//true*

        System.out.**println**(3>=4);*//false*

        System.out.**println**(3!=4);*//true*

    }

tips：如果进行多次判断，不能连着写。例如程序当中 ~~1 < x < 3~~是错误的。

### 第四节 逻辑运算符



逻辑运算符，是用来连接两个布尔类型结果的运算符，运算结果都是布尔值 true 或者 false

public static void **main**(String[] args)  {

        System.out.**println**(true && true);*//true*

        System.out.**println**(true && false);*//false*

        System.out.**println**(false && true);*//false，右边不计算*

        System.out.**println**(false || false);*//falase*

        System.out.**println**(false || true);*//true*

        System.out.**println**(true || false);*//true，右边不计算*

        System.out.**println**(!false);*//true*

    }

### 第五节 三元运算符

三元运算符首先判断条件是否成立，如果成立为true，那么将表达式A的值赋值给左侧的变量；如果不成立为false，那么将表达式B的值赋值给左侧的变量；二者选且必选其一。

使用格式如下：

数据类型 变量名 = **布尔类型表达式**？结果1：结果2

tips：

1. 必须同时保证表达式A和表达式B都符合左侧数据类型的要求。

2. 三元运算符的结果必须被使用。

public static void **main**(String[] args) {

    int a = 10; int b = 20;

    int max = a > b ? a : b; *// 最大值的变量*

    System.out.**println**("最大值：" + max); *// 20*

*// int result = 3 > 4 ? 2.5 : 10; // 错误写法1*

*// a > b ? a : b; // 错误写法2！三元运算符的结果必须被使用*

System.out.**println**(a > b ? a : b); *// 正确写法！*

}

## 第五章 流程控制

### 第一节 顺序结构与流程控制

程序的执行结果与执行流程息息相关，因此我们必须清楚每条语句的执行流程。最简单的流程是顺序结构，即根据编写的顺序，从上到下运行。

### 第二节 判断语句和选择语句

如果我们希望程序在某个时候根据不同的情况来执行不同的代码，那么我们将需要用到判断语句或选择语句。判断语句有三种格式，分别为：

* **if**

if(关系表达式)｛

   语句体;

｝

* **if...else**

if(关系表达式) {

       语句体1;

    }else {

       语句体2;

    }

* **if...else if ...else**

if (判断条件1) {

       执行语句1;

    } else if (判断条件2) {

       执行语句2;

    }

    ...

    }else if (判断条件n) {

      执行语句n;

    } else {

       执行语句n+1;

    }

* **if 语句和三元运算符的互换**

在某些简单的应用中，if语句是可以和三元运算符互换使用的。如下示例：

public static void **main**(String[] args) {

        int a = 10; int b = 20;

*//定义变量，保存a和b的较大值*

        int c;

        if(a > b) {

           c = a;

        } else {

           c = b;

        }

*//可以上述功能改写为三元运算符形式*

        c = a > b ? a:b;

    }

* **选择语句—switch**

除了判断语句外，一般对与多路选择分支，我们采用switch语句，使用格式如下：

switch(表达式) {

      case 常量值1:

        语句体1;

        break;

      case 常量值2:

        语句体2;

        break;

      ...

      default:

        语句体n+1;

        break;

}

tips：

1) 多个case后面的数值不可以重复。

2) switch后面小括号当中只能是下列数据类型：

基本数据类型：byte/short/char/int

引用数据类型：**String字符串(jdk7)**、enum枚举

3) switch语句格式可以很灵活：前后顺序可以颠倒，而且break语句还可以省略。

* **case 的穿透性**

在switch语句中，如果case的后面不写break，将出现穿透现象，也就是不会在判断下一个case的值，直接向后运行，直到遇到break，或者整体switch结束。

public static void **main**(String[] args) {

      int i = 5;

      switch (i){

        case 0:

          System.out.**println**("执行case0");

          break;

        case 5:

          System.out.**println**("执行case5");

        case 10:

          System.out.**println**("执行case10");

        default:

          System.out.**println**("执行default");

      }

    }

### 第三节 循环语句

如果我们需要运行一些重复的代码，那么可以使用循环语句来进行，循环语句有四要素如下：

* **循环结构四要素**

循环结构的基本组成部分，一般可以分成四部分：

1) 初始化语句：在循环开始最初执行，而且只做唯一一次。

2) 条件判断：如果成立，则循环继续；如果不成立，则循环退出。

3) 循环体：重复要做的事情内容，若干行语句。

4) 步进语句：每次循环之后都要进行的扫尾工作，每次循环结束之后都要执行一次。

* **循环语句1—for循环**

for(初始化表达式①; 布尔表达式②; 步进表达式④){

    循环体③

    }

* **循环语句2—while循环**

初始化表达式①

  while(布尔表达式②){

    循环体③

    步进表达式④

}

* **循环语句3--do...while**

初始化表达式①

    do{

    循环体③

    步进表达式④

}while(布尔表达式②);

Tips：do-while循环至少会被执行一次,可以用do{}while(flase)来实现goto语句。

* **死循环**

**死循环**就是循环中的条件永远为true，一般情况下不会结束的循环。例如：while(true){}。死循环用途良多，如在等待用户的输入时，一直死循环直到用户输入了正确的数值为止。

* **循环嵌套**

所谓循环嵌套 ，是指一个循环的循环体是另一个循环。比如for循环里面还有一个for循环，就是嵌套循环。总共的循环次数=外循环次数\*内循环次数

for(初始化表达式①; 循环条件②; 步进表达式⑦) {

        for(初始化表达式③; 循环条件④; 步进表达式⑥) {

           执行语句⑤;

        }

    }

### 第四节 跳出语句

如果希望对既定的流程进行干预，可以使用break或continue来跳出语句。

* **break**

break语句有两种用法：

1) 可以用在switch语句当中，一旦执行，整个switch语句立刻结束。

2) 还可以用在循环语句当中，一旦执行，整个循环语句立刻结束。打断循环。

public static void **main**(String[] args) {

        for (int i = 1; i<=10; i++) {

*//需求:打印完两次HelloWorld之后结束循环*

            if(i == 3){

              break;

            }

            System.out.**println**("HelloWorld"+i);

        }

    }

* **continue**

continue语句只能用在循环中，用来跳过本次循环continue语句之后的剩余内容，马上开始开一次的循环。**(注意不要把步进语句给跳过了！)**

public static void **main**(String[] args) {

        for (int i = 1; i <= 10; i++) {

*//需求:不打印第三次HelloWorld*

            if(i == 3){

              continue;

            }

            System.out.**println**("HelloWorld"+i);

        }

    }

## 第六章 数组

为什么要有数组呢？假如我们要存储全班50名学生的学号，显然定义50个int变量是很愚蠢的行为，那么有没有一种数据结构，能将这50个变量统一进行管理呢，答案是有的，在java中，最简单的就是用数组来实现，**数组就是存储多个相同数据类型数据的容器**。

### 第一节 数组的定义和访问

* **定义方式1：显示指定长度**

数组存储的数据类型[] 数组名字 = new 数组存储的数据类型[长度];

例：int[] arr = new int[3]; //定义存储三个整数的数组容器

* **定义方式2：显示指定内容**

数据类型[] 数组名 = new 数据类型[]{元素1,元素2,元素3...};

例：int[] arr = new int[]{1,2,3}; //定义存储1,2,3整数的容器

* **定义方式3：显示指定内容的省略格式**

数据类型[] 数组名 = {元素1,元素2,元素3...};

例：int[] arr = {1,2,3}; //定义存储1,2,3整数的容器

* **数组的初始化**

如果是使用**显示指定长度的方式**来定义的数组，那么其中的元素将会自动拥有一个默认值。规则符合堆内存初始化规则。

其实，即使是使用**显示指定内容的方式**来定义数组，初始化其实也有默认值的过程，只不过系统自动马上将默认值替换成为了大括号当中的具体数值。

* **数组的访问：索引**

**索引**(index)**是**每一个存储到数组的元素，都会自动的拥有一个从0开始的编号，我们可以通过数组的索引访问到数组中的元素。

public static void **main**(String[] args) {

*//定义存储int类型数组，赋值元素1，2，3，4，5*

        int[] arr = {1,2,3,4,5};

*//为0索引元素赋值为6*

        arr[0] = 6;

*//获取数组0索引上的元素*

        int i = arr[0];

        System.out.**println**(i);

*//直接输出数组0索引元素*

        System.out.**println**(arr[0]);

    }

Tips：与其它语言一样，java中的数组也是固定长度的，可以通过**.length**来获取数组长度。

获取数组的长度示例如下：

public static void **main**(String[] args) {

       int[] arr = new int[]{1,2,3,4,5};

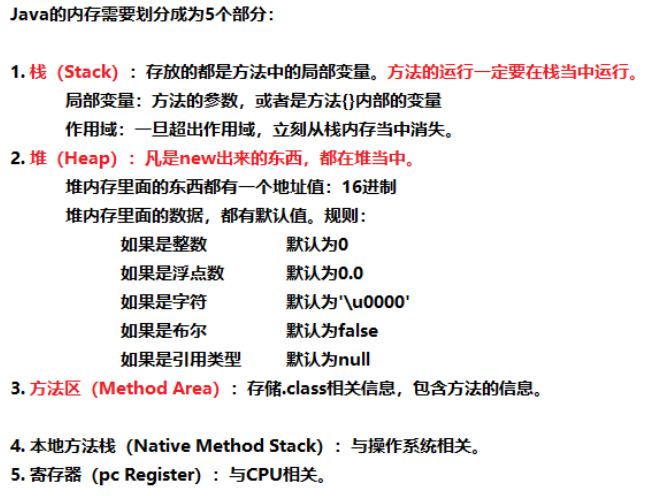
       System.out.**println**(arr.length);// 5

    }

### 第二节 数组的原理内存图

#### 1. Java 虚拟机的内存划分

Java虚拟机要运行程序，必须先将程序加载到内存，而为了提高内存空间的利用效率，因此对内存进行空间的分配和管理,每一片区域都有特定的处理数据方式和内存管理方式。



#### 2. 数组的内存图

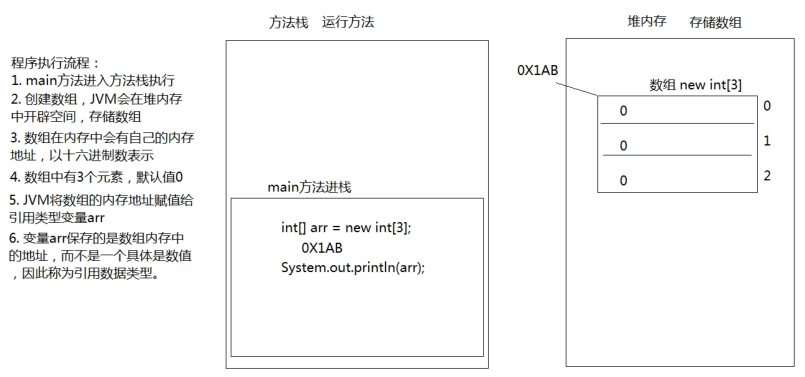
下面代码定义了一个数组，在内存中的存储如图所示：

public static void **main**(String[] args) {

       int[] arr = new int[3];

       System.out.**println**(arr);*//[I@5f150435*

    }



执行以上方法，将会输出的结果是[I@5f150435，这个是什么呢？就是数组在内存中堆地址的哈希值。new出来的内容，都是在堆内存中存储的，而方法中的变量arr保存的是数组的地址。如果想输入数组中的元素，应当采用索引的方式，如arr[0]。

#### 3. 两个变量指向同一个数组

如果存在两个数组变量，指向了同一个new出来的数组，那么会怎么样呢？

public static void **main**(String[] args) {

*// 定义数组，存储3个元素*

        int[] arr = new int[3];

*//数组索引进行赋值*

        arr[0] = 5;

        arr[1] = 6;

        arr[2] = 7;

*//输出3个索引上的元素值*

        System.out.**println**(arr[0]);

        System.out.**println**(arr[1]);

        System.out.**println**(arr[2]);

*//定义数组变量arr2，将arr的地址赋值给arr2*

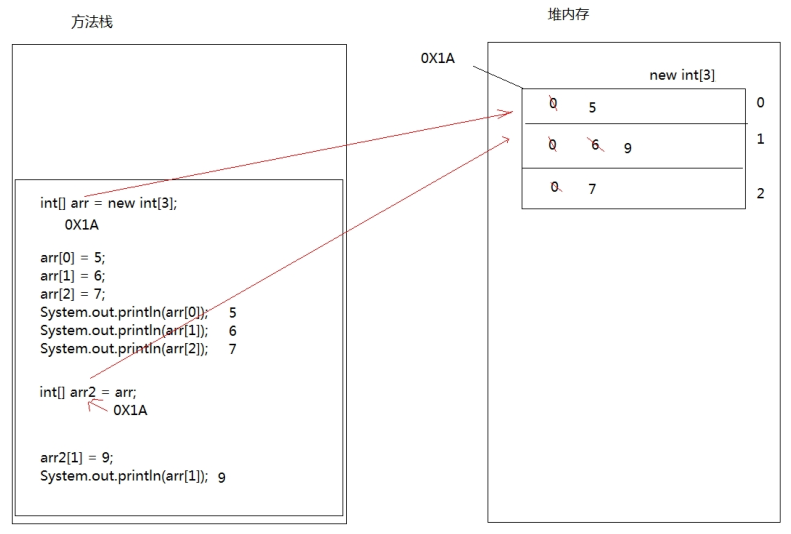
        int[] arr2 = arr;

        arr2[1] = 9;

        System.out.**println**(arr[1]);

}

我们猜测，这两个数组变量都是arr变量new出来数组的引用，都可以对该数组进行操作。因此以上代码中arr2的修改就是对arr的修改。实际上，内存分布也是如此。



tips：在java中，数组的类型**不带长度**，并且**不是一个”const指针”**，这两点与C/C++的数组不同。

### 第三节 数组的常见操作

#### 1. 数组作为方法的参数或返回值

数组是一种引用类型，作为方法的参数时，传递的是数组内存的地址。如下程序：

public static void **main**(String[] args) {

        int[] arr = {1,3,5};

        System.out.**println**(arr[0]);

**change**(arr);

        System.out.**println**(arr[0]);

}

    public static void **change**(int[] arr) {

       arr[0] = 200;

}

同样的道理，数组作为方法的返回值，返回的也是一个内存地址。

public static void **main**(String[] args) {

        int[] arr = **getArray**();  *//调用方法，接收到的是数组的内存地址*

        for (int i = 0; i < arr.length; i++) {

          System.out.**println**(arr[i]);

        }

    }

public static int[] **getArray**() {

    int[] arr = { 1, 3, 5, 7, 9 };

    return arr; *//返回数组的地址，返回到调用者*

}

#### 2. 数组的常见异常

* **数组越界异常**

观察以下代码，我们发现数组arr的最大索引是2，但缺想要打印索引为3的元素，那么在程序运行后，就会抛出 **ArrayIndexOutOfBoundsException** 异常。

public static void **main**(String[] args) {

        int[] arr = {1,2,3};

        System.out.**println**(arr[3]);

}

* **数组空指针异常**

观察以下代码，我们发现，在打印arr的第0个元素前，将arr指向了null，因此在索引时，就会出现**NullPointerException**指针异常。这是因为arr已经不保存任何数组的地址了。

public static void **main**(String[] args) {

        int[] arr = {1,2,3};

        arr = null;

        System.out.**println**(arr[0]);

}

## 第七章 方法

**方法**就是将一个功能抽取出来，把代码单独定义在一个大括号内，形成一个单独的功能。当我们需要这个功能的时候，就可以去调用。这样即实现了代码的复用性，也解决了代码冗余的现象。

#### 第一节 方法的定义与调用

对我们来说，一般方法有三个重要的元素，分别是：**返回值类型**、**方法名**、**参数列表**，特别的有些方法还拥有多个修饰符，如*public、static*等。方法定义的格式如下：

修饰符 返回值类型 **方法名** （参数列表）｛

     代码...

    return ;

｝

* **在定义方法时的一些注意事项**

1) 方法必须定义在一类中方法外, 不能定义在另一个方法的里面

public class **Demo** {

        public static void **main**(String[] args){

        }

*//正确写法，类中，main方法外面可以定义方法*

        public static void **method**(){}

    }

public class **Demo** {

    public static void **main**(String[] args){

*//错误写法，一个方法不能定义在另一方法内部*

        public static void **method**(){}

    }

}

2) 返回值类型，必须要和 return 语句返回的类型相同，否则编译失败 。

*// 返回值类型要求是int*

public static int **getSum**() {

        return 5;*// 正确，int类型*

        return 1.2;*// 错误，类型不匹配*

        return true;*// 错误，类型不匹配*

    }

3) 不能在 return 后面写代码， return 意味着方法结束，所有后面的代码永远不会执行，属于无效代码。

public static int **getSum**(int a,int b) {

       return a + b;

       System.out.**println**("Hello");*// 错误，return已经结束，这里不会执行，无效代码*

    }

* **方法的调用**

方法在定义完毕后，方法不会自己运行，必须被调用才能执行，我们可以在主方法main中来调用我们自己定义好的方法。在主方法中，直接写要调用的方法名字就可以调用了。

public static void **main**(String[] args) {

*//调用定义的方法method*

**method**();

    }

*//定义方法，被main方法调用*

    public static void **method**() {

       System.out.**println**("自己定义的方法，需要被main调用运行");

    }

#### 第二节 方法的重载

**方法重载**是指在同一个类中，允许存在一个以上的同名方法，只要它们的参数列表不同即可。

参数列表不同可以是**参数个数不同**、**参数类型不同**和**参数的顺序不同**三种形式。

tips:方法的重载与参数的名称以及返回值类型无关。

练习：判断哪些方法是重载关系。

public static void **open**(){} *// 正确重载*

public static void **open**(int a){} *// 正确重载*

static void **open**(int a,int b){} *// 代码错误：和第8行冲突*

public static void **open**(double a,int b){} *// 正确重载*

public static void **open**(int a,double b){} *// 代码错误：和第6行冲突*

public void **open**(int i,double d){} *// 代码错误：和第5行冲突*

public static void **OPEN**(){} *// 代码正确不会报错，但是并不是有效重载*

public static void **open**(int i,int j){} *// 代码错误：和第3行冲突*

# 第二篇 Java面向对象

## 第八章 面向对象和封装

Java语言是一种完全面向对象的程序设计语言。这里的**对象**泛指现实中一切事物，每种事物都具备自己的**属性和行为**。面向对象思想就是在计算机程序设计过程中，参照现实中事物，将事物的属性特征、行为特征抽象出来，描述成计算机事件的设计思想。 它区别于面向过程思想，强调的是通过调用对象的行为来实现功能，而不是自己一步一步的去操作实现。面向对象思想是一种更符合我们思考习惯的思想，它可以将复杂的事情简单化，并将我们从执行者变成了指挥者。

面向对象的语言中，包含了三大基本特征，即**封装、继承和多态**。

### 第一节 类和对象

#### 1. 类和对象的概念

* **什么是类**

**类是一组相关属性和行为的集合**。可以看成是一类事物的模板，使用事物的属性特征和行为特征来描述该类事物。如猫是一个类，属性有字、体重、年龄、颜色等，行为有走、跑、叫等。

* **什么是对象**

**对象是一类事物的具体体现**。对象是类的一个实例，必然具备该类事物的属性和行为。如某只猫是一个对象，它的属性是tom、5kg、2 years、yellow，行为有溜墙根走、蹦跶的跑、喵喵叫等。

* **类和对象的关系**

类是对一类事物的描述，是**抽象**的。对象是一类事物的实例，是**具体**的。类是对象的模板，对象是类的实体 。

#### 2. 类定义及使用

* **类的定义**

现实生活中的事务有属性和行为，而java类中采用**成员变量**和**成员方法**分别对应。定义一个java类的例子如下：

public class **Student** {

*//****成员变量 ：一定要定义在方法外面***

    String name；

    int age；

*//****成员方法：去掉static关键字***

    public void **study**() {

        System.out.**println**("好好学习，天天向上");

    }

}

* **类的使用**

要想使用这个java类，必须先定义一个该类的对象，对象的定义格式如下：

类名 对象名 = new 类名();

定义好对象之后，就可以通过点运算符来使用它的成员变量和成员方法了，下面是一个创建对象来使用student类的例子：

public class **Test01\_Student** {

      public static void **main**(String[] args) {

*//创建对象格式：类名 对象名 = new 类名();*

        Student s = new **Student**();

        System.out.**println**("s:"+s); *//cn.itcast.Student@100363*

*//直接输出成员变量值*

        System.out.**println**("姓名："+s.name); *//null*

        System.out.**println**("年龄："+s.age); *//0*

        System.out.**println**("‐‐‐‐‐‐‐‐‐‐");

*//给成员变量赋值*

        s.name = "赵丽颖";

        s.age = 18;

*//再次输出成员变量的值*

        System.out.**println**("姓名："+s.name); *//赵丽颖*

        System.out.**println**("年龄："+s.age); *//18*

        System.out.**println**("‐‐‐‐‐‐‐‐‐‐");

*//调用成员方法*

        s.**study**(); *// "好好学习，天天向上"*

      }

}

tips：

1) 在java中，打印引用类型变量名出来的是一个内存地址哈希值。

2) 在java中，所有引用类型变量都是在堆内存new出来的，并且会被初始化为默认值。

* **匿名对象**

匿名对象指只有创建对象的语句，但并没有将该对象赋值给某个变量保存，这种对象在创建时直接调用方法，并且只能使用一次，示例如下：

new **Scanner**(System.in).**nextInt**();

### 第二节 类的封装

在编程语言中，广义的封装就是将一些细节信息隐藏起来，对于外界不可见。具体到java类的封装，就是将属性隐藏起来，若需要访问某个属性，提供公共方法对其访问。

#### 1. 封装的实现方式-private

private是java的一个权限修饰符，可以修饰成员变量或成员方法，它代表访问的最小权限，即只有可以在本类中进行访问。一个使用private修饰成员变量的例子如下：

public class **Student** {

      private String name;

      private int age;

}

如果我们对成员变量使用了private关键字进行修饰，那么外部将无法访问该成员变量，我们可以根据需要，定义一些**get/set方法**来实现间接操作。get/set方法可以通过**Alt+Insert键**来自动生成，并且方法名是小驼峰式命名法。下面是成员变量name的get/set方法：

public void **setName**(String n) {

    name = n;

}

public String **getName**() {

    return name;

}

tips：特殊的，对于布尔型成员变量，其get方法名应写成isXxx，而不是getXxx。

public boolean **isMale**() {

    return male;

}

#### 2. 封装的优化-this

在成员方法中，如果我们定义了一个形参，刚好与已有的成员变量重名，那么该形参会将成员变量给隐藏掉，如果要继续使用成员变量，就需要使用this关键字来显示指明。this代表了当前对象的地址值，方法被哪个对象调用，方法中的this就代表那个对象。下面是一个this的使用示例：

public class **Student** {

      private String name;

      public void **setName**(String name) {

*//name = name;*

        this.name = name;

      }

      public String **getName**() {

        return name;

      }

}

tips： 在static方法是没有this的，因为调用它不需要任何对象。

#### 3. 封装的优化-构造器

在new出一个对象时，虽然会被初始化为默认值，但这些默认值并不一定是我们想要的，我们可以通过构造器来对对象来进行初始化，构造器的代码也可以通过**Alt+Insert键**进行生成，下面是studet类的两个构造方法：

public class **Student** {

      private String name;

      private int age;

*// 无参数构造方法*

      public **Student**() {}

*// 全参数构造方法*

      public **Student**(String name,int age) {

        this.name = name;

        this.age = age;

      }

}

构造方法有几点需要明白：

1) 如果你类中没有任何的构造器，那么编译器会默认生成一个空实现无参构造。

2) 构造方法的方法名一定要与类名相同，且构造方法是没有返回值类型的。

#### 4. JavaBean

**JavaBean**是Java的一种语言编写类规范。符合 JavaBean 的类，一般满足下面四点：

*1) 所有的成员变量都要使用private关键字修饰  
2) 为每一个成员变量编写一对儿Getter/Setter方法  
3) 编写一个无参数的构造方法  
4) 编写一个全参数的构造方法*

### 第三节 static关键字

**static**可以用来修饰的成员变量和成员方法，被修饰的成员是属于类的，而不是单单是属

于某个对象的。也就是说，既然属于类，就可以不靠创建对象来调用了。

#### 1. static修饰成员变量

当**static**修饰成员变量时，该变量称为**类变量**。该类的每个对象都**共享**同一个类变量的值。任何对象都可以更改该类变量的值，但也可以在不创建该类的对象的情况下对类变量进行操作。如下例，通过类变量来生成学生编号：

public class **Student** {

      private String name;

      private int age;

      private int sid;

*// 类变量，记录学生数量，分配学号*

      public static int numberOfStudent = 0;

      public **Student**(String name, int age){

        this.name = name;

        this.age = age;

*// 通过 numberOfStudent 给学生分配学号*

        this.sid = ++numberOfStudent;

      }

*// 打印属性值*

      public void **show**() {

        System.out.**println**("Student : name=" + name + ", age=" + age + ", sid=" + sid );

      }

}

#### 2. static修饰成员方法

当 **static** 修饰成员方法时，该方法称为**类方法/静态方法** 。静态方法在声明中有 static ，建议使用类名来调用，而不需要创建类的对象。在Student类中加上下面静态方法：

public static void **showNum**() {

      System.out.**println**("num:" +  numberOfStudent);

}

tips：

1) 虽然也可以通过对象来调用静态方法，但不推荐这样使用。

2) 静态方法属于类，没有this关键字，因此**不能直接访问普通成员变量或成员方法**

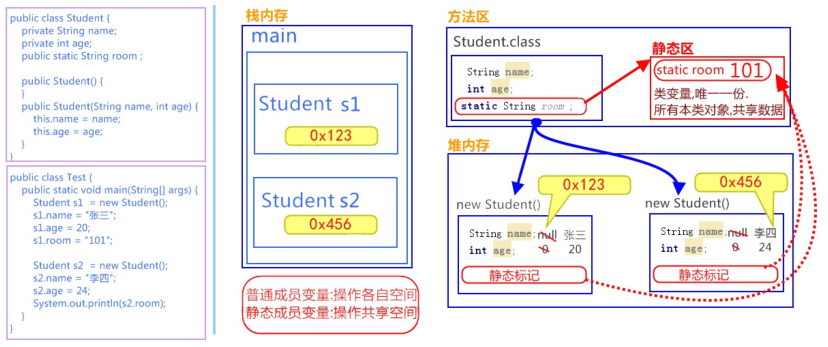
#### 3. static与静态区

被static修饰的内容，具有以下特征：

1) 随着类的加载而加载的，且只加载一次。

2) 存储于一块固定的内存区域（静态区），所以，可以直接被类名调用。

3) 它优先于对象存在，所以，可以被所有对象共享。



#### 4. static修饰代码块

**静态代码块**是定义在类中，但是在方法外的一段被static修饰的代码，它随着类的加载而执行且执行一次，优先于 main方法和构造方法的执行，一般用来给类变量进行初始化。

public class **Game** {

      public static int number;

      public static ArrayList<String> list;

      static {

*// 给类变量赋值*

        number = 2;

        list = new ArrayList<String>();

*// 添加元素到集合中*

        list.**add**("张三");

        list.**add**("李四");

      }

}

### 第四节 常用的java类

Java语言是一门成熟的语言，对许多常用的功能封装成了Java类，我们可以查看 **JDK\_API\_1\_6\_zh\_CN.CHM** 来使用，下面我们将简要介绍最常用的几个Java类。

#### 1. Scanner类

Scanner类是一个可以使用正则表达式来解析基本类型和字符串的简单文本扫描器。使用它可以分为三个步骤：

* **导包**

除使用java.lang包外，若想使用其它包中的类，都需要进行导包操作，Scanner类导包语句如下：

import java.util.Scanner; // import 包名.类名;

* **定义对象**

如果要想使用Sanner类的成员方法，必须先创建该类的对象，如下：

Scanner sc = new **Scanner**(System.in);

System.in是Scanner类构造函数的参数，在现阶段，我们只需要知道这代表从键盘输入即可。

* **使用方法**

要想实现某个功能，必须调用相应的方法，Scanner类的方法有很多，如下例：

*//1. 导包*

import java.util.Scanner;

public class **Demo01\_Scanner** {

   public static void **main**(String[] args) {

*//2. 创建键盘录入数据的对象*

     Scanner sc = new **Scanner**(System.in);

*//3. 接收数据*

     System.out.**println**("请录入一个整数：");

     int i = sc.**nextInt**();

*//4. 输出数据*

     System.out.**println**("i:"+i);

   }

}

#### 2. Random类

Random类的实例用于生成伪随机数流。下面是一个生成范围1-n随机数的示例：

*// 导包*

import java.util.Random;

public class **Test01Random** {

  public static void **main**(String[] args) {

int n = 50;

*// 创建对象*

Random r = new **Random**();

*// 获取随机数*

int number = r.**nextInt**(n) + 1;

*// 输出随机数*

    System.out.**println**("number:" + number);

  }

}

Tips

1) 如果用相同的种子创建两个 Random 实例，则对每个实例进行相同的方法调用序列，它们将生成并返回相同的数字序列。

2) 多应用程序会发现 **Math.random()** 方法更易于使用。

#### 3. ArrayList类

ArrayList是一个**大小可变的数组**类。使用方法如下：

public class **Demo01ArrayListMethod** {

      public static void **main**(String[] args) {

*//创建集合对象*

        ArrayList<String> list = new ArrayList<String>();//第二个String在JDK7后可省略

*//添加元素*

        list.**add**("hello");

        list.**add**("world");

        list.**add**("java");

//对于ArrayList对象来说，直接打印的不是一个地址值

System.out.**println**(list);  *// [hello, world, java]*

*//public E get(int index):返回指定索引处的元素*

        System.out.**println**("get:"+list.**get**(0));

        System.out.**println**("get:"+list.**get**(1));

        System.out.**println**("get:"+list.**get**(2));

*//public int size():返回集合中的元素的个数*

        System.out.**println**("size:"+list.**size**());

*//public E remove(int index):删除指定索引处的元素，返回被删除的元素*

        System.out.**println**("remove:"+list.**remove**(0));

*//遍历输出*

        for(int i = 0; i < list.**size**(); i++){

          System.out.**println**(list.**get**(i));

        }

      }

}

tips：ArrayList中模板参数只能是引用类型，如果要使用基本类型，必须使用其包装类型，一般将基本类型的首字母大写即可，但特别的，int类型 -> Integer，char类型 -> Character。

ArrayList<Integer> list = new ArrayList<>();

#### 4. String类

**java.lang.String 类**代表字符串。Java程序中所有的字符串文字（例如 "abc" ）都可以被看作是实现此类的实例。它有几个特点如下：

1) **字符串不变：字符串的值在创建后不能被更改。**

String s1 = "abc";

s1 += "d";

System.out.**println**(s1); *// "abcd"*

*// 内存中有"abc"，"abcd"两个对象，s1从指向"abc"，改变指向，指向了"abcd"。*

2) 因为String对象是不可变的，所以它们可以被共享。

String s1 = "abc";

String s2 = "abc";

*// 内存中只有一个"abc"对象被创建，同时被s1和s2共享。*

3) 字符串效果上相当于是char[]字符数组，但是底层原理是byte[]字节数组。

例如：

String str = "abc";

相当于：

char data[] = {'a', 'b', 'c'};

String str = new **String**(data);

*// String底层是靠字符数组实现的。*

* **String类的构造**

String类的构造可以有很多种，下面介绍常见的几种：

*// 无参构造*

String str = new String();

*// 通过字符数组构造*

char chars[] = {'a', 'b', 'c'};

String str2 = new **String**(chars);

*// 通过字节数组构造*

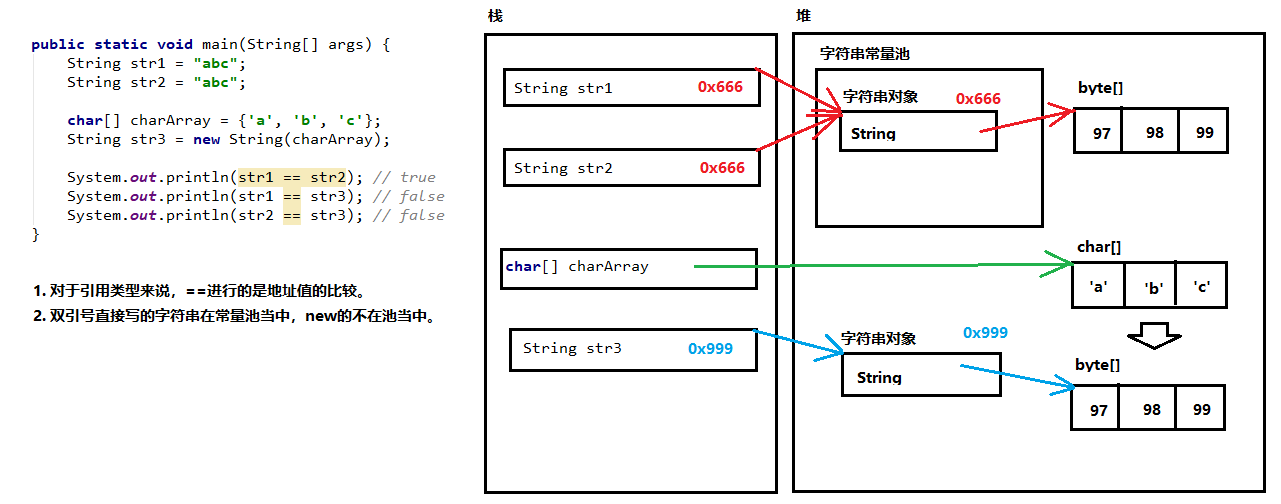
byte bytes[] = { 97, 98, 99 };

String str3 = new **String**(bytes);

tips：在代码中写的字符串字面量也是一个String类对象。

* **字符串常量池**

字符串常量池：程序当中直接写上的双引号字符串，就在字符串常量池中。



* **判断功能的方法**

public class **String\_Demo01** {

      public static void **main**(String[] args) {

*// 创建字符串对象*

        String s1 = "hello";

        String s2 = "hello";

        String s3 = "HELLO";

*// boolean equals(Object obj):比较字符串的内容是否相同*

        System.out.**println**(s1.**equals**(s2)); *// true*

        System.out.**println**(s1.**equals**(s3)); *// false*

        System.out.**println**("‐‐‐‐‐‐‐‐‐‐‐");

*//boolean equalsIgnoreCase(String str):比较字符串的内容是否相同,忽略大小写*

        System.out.**println**(s1.**equalsIgnoreCase**(s2)); *// true*

        System.out.**println**(s1.**equalsIgnoreCase**(s3)); *// true*

        System.out.**println**("‐‐‐‐‐‐‐‐‐‐‐");

      }

}

tips：如果比较双方一个常量一个变量，推荐把常量字符串写在前面。推荐："abc".equals(str)。

* **获取功能的方法**

public class **String\_Demo02** {

  public static void **main**(String[] args) {

*//创建字符串对象*

String s = "helloworld";

*// int length():获取字符串的长度，其实也就是字符个数*

    System.out.**println**(s.**length**());

System.out.**println**("‐‐‐‐‐‐‐‐");

*// String concat (String str):将将指定的字符串连接到该字符串的末尾.*

    String s = "helloworld";

    String s2 = s.**concat**("\*\*hello itheima");

     System.out.**println**(s2);*// helloworld\*\*hello itheima*

*// char charAt(int index):获取指定索引处的字符*

    System.out.**println**(s.**charAt**(0));

    System.out.**println**(s.**charAt**(1));

System.out.**println**("‐‐‐‐‐‐‐‐");

*// int indexOf(String str):获取str在字符串对象中第一次出现的索引,没有返回‐1*

    System.out.**println**(s.**indexOf**("l"));

    System.out.**println**(s.**indexOf**("owo"));

    System.out.**println**(s.**indexOf**("ak"));

System.out.**println**("‐‐‐‐‐‐‐‐");

*// String substring(int start):从start开始截取字符串到字符串结尾*

    System.out.**println**(s.**substring**(0));

    System.out.**println**(s.**substring**(5));

System.out.**println**("‐‐‐‐‐‐‐‐");

*// String substring(int start,int end):从start到end截取字符串。含start，不含end。*

    System.out.**println**(s.**substring**(0, s.**length**()));

    System.out.**println**(s.**substring**(3,8));

  }

}

* **转换功能的方法**

public class **String\_Demo03** {

  public static void **main**(String[] args) {

*//创建字符串对象*

String s = "abcde";

*// char[] toCharArray():把字符串转换为字符数组*

    char[] chs = s.**toCharArray**();

    for(int x = 0; x < chs.length; x++) {

      System.out.**println**(chs[x]);

    }

System.out.**println**("‐‐‐‐‐‐‐‐‐‐‐");

*// byte[] getBytes ():把字符串转换为字节数组*

    byte[] bytes = s.**getBytes**();

    for(int x = 0; x < bytes.length; x++) {

      System.out.**println**(bytes[x]);

    }

System.out.**println**("‐‐‐‐‐‐‐‐‐‐‐");

*// 替换字母it为大写IT*

    String str = "itcast itheima";

    String replace = str.**replace**("it", "IT");

    System.out.**println**(replace); *// ITcast ITheima*

    System.out.**println**("‐‐‐‐‐‐‐‐‐‐‐");

  }

}

* **分割功能的方法**

public class **String\_Demo03** {

      public static void **main**(String[] args) {

*//创建字符串对象*

        String s = "aa|bb|cc";

*//public String[] split(String regex)*

        String[] strArray = s.**split**("|"); *// ["aa","bb","cc"]*

        for(int x = 0; x < strArray.length; x++) {

          System.out.**println**(strArray[x]); *// aa bb cc*

        }

      }

}

tips：如果按照英文句点“.”进行切分，必须写"\\."（两个反斜杠）

#### 5. Arrays类

**java.util.Arrays** 此类包含用来操作数组的各种方法，比如**排序**和搜索等。其所有方法均为静态方法，调用起来非常简单。

* **数组的字符串表示**

public static void **main**(String[] args) {

*// 定义int 数组*

      int[] arr  =  {2,34,35,4,657,8,69,9};

*// 打印数组,输出地址值*

      System.out.**println**(arr); *// [I@2ac1fdc4*

*// public static String toString(int[] a) ：返回指定数组内容的字符串表示形式*

      String s = Arrays.**toString**(arr);

*// 打印字符串,输出内容*

      System.out.**println**(s); *// [2, 34, 35, 4, 657, 8, 69, 9]*

}

* **数组排序**

public static void **main**(String[] args) {

*// 定义int 数组*

      int[] arr  =  {24, 7, 5, 48, 4, 46, 35, 11, 6, 2};

*// public static void sort(int[] a) ：对指定的 int 型数组按数字升序进行排序。*

      Arrays.**sort**(arr);

      System.out.**println**("排序后:"+ Arrays.**toString**(arr));*// 排序后:[2, 4, 5, 6, 7, 11, 24, 35, 46,48]*

}

#### 6. Math类

**java.lang.Math** 类包含用于执行基本数学运算的方法，如初等指数、对数、平方根和三角函数。类似这样的工具类，其所有方法均为静态方法，并且不会创建对象，调用起来非常简单。

* **取对值**

*// public static double abs(double a) 返回 double 值的绝对值。*

double d1 = Math.**abs**(‐5); *//d1的值为5*

double d2 = Math.**abs**(5); *//d2的值为5*

* **向上/向下取整**

*// public static double ceil(double a) ：返回大于等于参数的最小的整数。*

double d1 = Math.**ceil**(3.3); *//d1的值为 4.0*

double d2 = Math.**ceil**(‐3.3); *//d2的值为 ‐3.0*

double d3 = Math.**ceil**(5.1); *//d3的值为 6.0*

*// public static double floor(double a) ：返回小于等于参数最大的整数。*

double d1 = Math.**floor**(3.3); *//d1的值为3.0*

double d2 = Math.**floor**(‐3.3); *//d2的值为‐4.0*

double d3 = Math.**floor**(5.1); *//d3的值为 5.0*

* **四舍五入**

*// public static long round(double a) ：返回最接近参数的 long。(相当于四舍五入方法)*

long d1 = Math.**round**(5.5); *//d1的值为6.0*

long d2 = Math.**round**(5.4); *//d2的值为5.0*

tips: Math.PI代表近似的圆周率常量（double）

## 第九章 继承与多态

### 第一节 继承 extends

#### 1. 什么是继承

假如多个类具有相同的属性和行为，那么可以将这些相同点抽出来，单独作为一个类，这个类叫做**父类/超类/基类(SuperClass)**，这样不仅提高了代码之间的复用性，并且让类之间产生了关联(is-a关系)，这是多态产生的前提。

反过来，其它类从父类获取属性和行为的动作叫做继承，**继承**就是子类继承父类的属性和行为，使得子类对象具有与父类相同的属性、相同的行为。一个继承的例子如下：

class **Employee** {

    String name; *// 定义name属性*

*// 定义员工的工作方法*

    public void **work**() {

        System.out.**println**("尽心尽力地工作");

    }

}

class **Teacher** extends **Employee** {   *//定义讲师类Teacher 继承 员工类Employee*

*// 定义一个打印name的方法*

    public void **printName**() {

        System.out.**println**("name=" + name);

    }

}

那么Teacher类可以继承Employee类的所有属性和行为，使用Teacher类如下：

public class **ExtendDemo01** {

    public static void **main**(String[] args) {

*// 创建一个讲师类对象*

        Teacher t = new **Teacher**();

*// 为该员工类的name属性进行赋值*

        t.name = "小明";

*// 调用该员工的printName()方法*

        t.**printName**(); *// name = 小明*

*// 调用Teacher类继承来的work()方法*

        t.**work**();  *// 尽心尽力地工作*

    }

}

#### 2. 继承后的成员变量

如果子类成员变量与父类成员变量不重名，那么是可以直接访问的(父类的private变量除外)，没有任何影响。但如果出现了同名的成员变量时，在子类中需要访问父类中非私有成员变量时，需要使用 **super** 关键字，修饰父类成员变量，如：

public class **Zi** extends **Fu** {

    int num = 20;

    public void **method**() {

        int num = 30;

        System.out.**println**(num); *// 30，局部变量*

        System.out.**println**(this.num); *// 20，本类的成员变量*

        System.out.**println**(super.num); *// 10，父类的成员变量*

    }

}

#### 3. 继承后的成员方法[重写]

*对象调用方法时，会先在子类中查找有没有对应的方法，若子类中存在就会执行子类中的方法，若子类中不存在就会执行父类中相应的方法。*如果两者成员方法不重名，那么是没有任何影响的，但如果子类中出现重名的父类成员方法，这时的访问是一种特殊情况，叫做**方法重写 (Override)**。

**方法重写要求方法的三要素(返回值类型、方法名、参数列表)完全相同，并且权限不能小于父类的权限**，如果你不确定是否为正确重写，最好在重写方法前加上**@override**注解。

class **Phone** {

    public void **sendMessage**(){

        System.out.**println**("发短信");

    }

    public void **call**(){

        System.out.**println**("打电话");

    }

    public void **showNum**(){

        System.out.**println**("来电显示号码");

    }

}

class **NewPhone** extends **Phone** {

*//重写父类的来电显示号码功能，并增加自己的显示姓名和图片功能*

    @Override

    public void **showNum**(){

*//调用父类已经存在的功能使用super*

        super.**showNum**();

*//增加自己特有显示姓名和图片功能*

        System.out.**println**("显示来电姓名");

        System.out.**println**("显示头像");

    }

}

public class **ExtendsDemo06** {

    public static void **main**(String[] args) {

*// 创建子类对象*

           NewPhone np = new **NewPhone**()；

*// 调用父类继承而来的方法*

            np.**call**();

*// 调用子类重写的方法*

           np.**showNum**();

    }

}

tips：super关键字的用法有三种：

1) 在子类的成员方法中，访问父类的成员变量。

2) 在子类的成员方法中，访问父类的成员方法。

3) 在子类的构造方法中，访问父类的构造方法。

#### 4. 继承后的构造器

构造器是用来初始化刚创建的对象的，不属于类的属性或行为，因此子类不会继承。但子类对象构造前必须先构造出一个父类对象，示例代码如下：

class **Fu** {

**Fu**(){

      System.out.**println**("Fu()");

    }

}

class **Zi** extends **Fu** {

**Zi**(){

       super();//显示写出父类的构造器，且必须在第一行

       System.out.**println**("Zi()");

}

**Zi**(int n){

        this(); //在子类构造器中调用子类其它构造器

        this->n = n;

    }

}

public class **ExtendsDemo07**{

  public static void **main** (String args[]){

    Zi zi = new **Zi**();

  }

}

*/\**

*输出结果：*

*Fu（）*

*Zi（）*

*\*/*

tips:

1) 子类构造器的第一行必须为父类的构造器，即使不显示写出，编译器也会自动生成super()。

2) 在子类构造器中，可以在第一行调用本类的其它构造器，但super() 和 this()不能同时出现。

#### 5. Java继承的特点

1) Java只支持单继承，不支持多继承

*//一个类只能有一个父类，不可以有多个父类。*

class **C** extends **A**{}  *//ok*

class **C** extends **A**，**B**... *//error*

2) Java支持多层继承(继承体系)。

class **A**{}

class **B** extends **A**{}

class **C** extends **B**{}

tips: Java中顶层父类是Object类。所有的类默认继承Object，作为父类。

3) 子类和父类是一种相对的概念。

### 第二节 抽象类 abstract

#### 1. 抽象类的概念

父类中的方法，被它的子类们重写，子类各自的实现都不尽相同。那么父类的方法声明和方法主体，只有声明还有意义，而方法主体则没有存在的意义了。我们把没有方法主体的方法称为**抽象方法**。Java语法规定，包含抽象方法的类就是**抽象类**。

#### 2. 抽象类的定义和实现

如果一个类包含抽象方法，那么该类必须是抽象类。抽象类必须加上abstract修饰符，如下例：

public abstract class **Animal** {

    public abstract void **run**()；//没有主体的抽象方法

}

继承抽象类的子类必须重写父类所有的抽象方法，除非该子类也是成抽象类，这里的方法重写，是子类对父类抽象方法的完成实现，我们将这种方法重写的操作，也叫做**实现方法**。

public class **Cat** extends **Animal** {

@Override

    public void **run** (){

       System.out.**println**("小猫在墙头走~~~")；

    }

}

public class **CatTest** {

   public static void **main**(String[] args) {

*// 创建子类对象*

        Cat c = new **Cat**();

*// 调用run方法*

        c.**run**();

   }

}

*/\**

*输出结果：*

*小猫在墙头走~~~*

*\*/*

#### 3. 抽象类的几点注意事项

1) 抽象类不能创建对象，编译无法通过，只能创建其非抽象子类的对象。

理解：假设创建了抽象类的对象，调用抽象的方法，而抽象方法没有具体的方法体，没有意义。

2) 抽象类中，可以有构造方法，是供子类创建对象时，初始化父类成员使用的。

理解：子类的构造方法中，有默认的super()，需要访问父类构造方法。

3) 抽象类中，不一定包含抽象方法，但是有抽象方法的类必定是抽象类。

理解：未包含抽象方法的抽象类，目的就是不想让调用者创建该类对象，通常用于某些特殊的类结构设计。

4) 抽象类的子类，必须重写抽象父类中所有的抽象方法，除非该子类也是抽象类。

理解：假设不重写所有抽象方法，则类中可能包含抽象方法。那么创建对象后，调用抽象的方法，没有意义。

### 第三节 接口 interface

#### 1. 接口的概念

**接口**是Java语言中一种引用类型，是方法的集合，内部主要就是封装了方法，包含**抽象方法**（JDK 7及以前），**默认方法**和**静态方法**（JDK 8）以及**私有方法**（JDK 9）。

#### 2. 接口的定义

接口的定义，它与定义类方式相似，但是使用 **interface** 关键字。它也会被编译成.class文件，但一定要明确它并不是类，而是另外一种引用数据类型。

public interface **InterFaceName** {

*// 抽象方法：使用****public abstract****修饰，可以省略，没有方法体，非抽象子类必须实现该方法。*

    public abstract void **method**();

*//默认方法：使用  default 修饰，不可省略，供子类调用或者子类选择性重写。*

    public default void **method**() {

*// 执行语句*

    }

*//静态方法：使用  static 修饰，供接口直接调用。*

    public static void **method2**() {

*// 执行语句*

    }

*//私有方法：使用  private 修饰，供接口中的默认方法或者静态方法调用。*

    public interface **InterFaceName** {

        private void **method**() {

*// 执行语句*

        }

    }

}

tips：

1) 接口中，无法定义成员变量，但是可以定义常量，其值不可以改变，默认使用 **public static final**修饰,修饰符可以省略。

2) 接口中，没有构造方法，不能创建对象。

3) 接口中，没有静态代码块。

public interface **MyInterfaceConst** {

*// 这其实就是一个常量，一旦赋值，不可以修改*

    public static final int NUM\_OF\_MY\_CLASS = 12; //必须进行初始化

}

#### 3. 接口的实现和使用 implements

接口不能创建对象，必须被实现后才能使用。实现的动作类似继承，格式相仿，只是关键字不同，实现使用**implements**关键字。一个非抽象类要想实现一个接口：

1) 对于接口的**抽象方法**：必须实现所有的抽象方法。

2) 对于接口的**默认方法**：可以沿用，也可以重写实现。

3) 对于接口的**静态方法**：只能通过接口名调用，不需要实现，也无法实现。

4) 对于接口的**私有方法**：接口的私有方法只能被接口的默认方法使用，不能被实现，如果该私有方法是静态的，那么还可以被接口的静态方法使用。

#### 4. 接口的多实现

在继承体系中，一个类只能继承一个父类。但对于接口而言，一个类是可以实现多个接口的，这叫做接口的多实现。并且，一个类可以在继承一个父类的同时，再实现多个接口。格式如下：

class **类名** [extends 父类名] implements 接口名1,接口名2,接口名3... {

*// 重写接口中抽象方法【必须】*

*// 重写接口中默认方法【不重名时可选】*

}

一个非抽象类实现多个接口时：

1) 对于**抽象方法**：必须重写所有接口的所有抽象方法，如果抽象方法有重名的，只需要重写一次。

2) 对于**默认方法**：可以沿用所有接口的所有默认方法，也可以选择性实现，但**如果出现了重名的默认方法，那么必须重写一次该方法**。

3) 对于**静态方法**：不可以实现任何接口的静态方法，即使重名也无所谓，因为静态方法是通过接口名来调用的。

4) 对于**私有方法**：私有方法作为默认方法和静态方法的辅助，互不干涉，也无需实现。

* **继承来的方法和接口的默认方法重名：**子类会优先使用继承来的方法，如下例：

interface **A** {

        public default void **methodA**(){

            System.out.**println**("AAAAAAAAAAAA");

        }

}

class **D** {

        public void **methodA**(){

            System.out.**println**("DDDDDDDDDDDD");

        }

}

class **C** extends **D** implements **A** {

*// 未重写methodA方法*

}

public class **Test** {

        public static void **main**(String[] args) {

            C c = new **C**();

            c.**methodA**();

        }

}

*/\**

*输出结果:*

*DDDDDDDDDDDD*

*\*/*

#### 5. 一个接口继承多个接口

接口也能继承其它接口，并且能同时继承多个，如果父接口中的默认方法有重名的，那么子接口需要重写一次。代码如下：

interface **A** {

    public default void **method**(){

        System.out.**println**("AAAAAAAAAAAAAAAAAAA");

    }

}

interface **B** {

    public default void **method**(){

        System.out.**println**("BBBBBBBBBBBBBBBBBBB");

    }

}

interface **D** extends **A**,**B**{

    @Override

    public default void **method**() {

        System.out.**println**("DDDDDDDDDDDDDD");

    }

}

tips：子接口在重写默认方法时，default关键字可以保留，而子类重写默认方法时，不可以保留。

### 第四节 多态

#### 1. 多态的概念

**多态**是继封装、继承之后，面向对象的第三大特性。是指同一行为，具有多个不同表现形式。多态必须满足三个条件：

1) 有继承或实现关系

2) 子类进行了方法的重写

3) 父类引用指向子类对象

#### 2. 多态的实现

多态必须使用父类的引用指向子类的对象，当在使用父类的引用调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，执行的是子类重写后方法。在查找时符合以下原则：

1) 对于成员变量：看等号左边是谁，优先用谁，没有则向上找。(编译看左边，运行还是看左边)

2) 对于成员方法：看new的是谁，就优先用谁，没有则向上找。(编译看左边，运行看右边)

public class **Demo02MultiMethod** {

    public static void **main**(String[] args) {

        Fu obj = new **Zi**(); *// 多态*

        obj.**method**(); *// 父子都有，优先用子*

        obj.**methodFu**(); *// 子类没有，父类有，向上找到父类*

*// 编译看左边，左边是Fu，Fu当中没有methodZi方法，所以编译报错。*

*//        obj.methodZi(); // 错误写法！*

    }

}

一个多态的例子如下：

public abstract class **Animal** {

    public abstract void **eat**();

}

class **Cat** extends **Animal** {

    public void **eat**() {

        System.out.**println**("吃鱼");

    }

}

class **Dog** extends **Animal** {

    public void **eat**() {

        System.out.**println**("吃骨头");

    }

}

public class **Test** {

    public static void **main**(String[] args) {

*// 父类引用指向子类对象Cat*

        Animal a1 = new **Cat**();

*// 调用的是 Cat 的 eat*

        a1.**eat**();

*// 父类引用指向子类对象Dog*

        Animal a2 = new **Dog**();

*// 调用的是 Dog 的 eat*

        a2.**eat**();

    }

}

#### 3. 多态的好处

先来看看下面一段代码，

public class **Test** {

    public static void **main**(String[] args) {

*// 多态形式，创建对象*

        Cat c = new **Cat**();

        Dog d = new **Dog**();

*// 调用showCatEat*

**showCatEat**(c);

*// 调用showDogEat*

**showDogEat**(d);

*//以上两个方法, 均可以被showAnimalEat(Animal a)方法所替代，而执行效果一致*

**showAnimalEat**(c);

**showAnimalEat**(d);

    }

    public static void **showCatEat** (Cat c){

        c.**eat**();

    }

    public static void **showDogEat** (Dog d){

        d.**eat**();

    }

    public static void **showAnimalEat** (Animal a){

        a.**eat**();

    }

}

由于多态特性的支持，showAnimalEat方法的Animal类型，是Cat和Dog的父类类型，父类类型接收子类对象，当然可以把Cat对象和Dog对象，传递给方法。当eat方法执行时，多态规定，执行的是子类重写的方法，那么效果自然与showCatEat、showDogEat方法一致，所以showAnimalEat完全可以替代以上两方法。

不仅仅是替代，在扩展性方面，无论之后再多的子类出现，我们都不需要编写showXxxEat方法了，直接使用showAnimalEat都可以完成。所以，多态的好处，体现在，可以使程序编写的更简单，并有良好的扩展。

#### 4. 引用类型转型

多态的转型分为向上转型与向下转型两种：

* **向上转型**

多态本身是子类类型向父类类型向上转换的过程，这个过程是默认的。当父类引用指向一个子类对象时，便是向上转型。格式如下：

父类类型  变量名 = new 子类类型();

如：Animal a = new **Cat**();

* **向下转型**

向下转型时父类类型向子类类型向下转换的过程，这个过程是强制的。一个已经向上转型的子类对象，将父类引用转为子类引用，可以使用强制类型转换的格式，便是向下转型。格式如下：

子类类型 变量名 = (子类类型) 父类变量名;

如:Cat c =(Cat) a;

* **为什么要转型**

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误。也就是说，不能调用子类拥有而父类没有的方法。编译都错误，更别说运行了。这也是多态给我们带来的一点"小麻烦"。所以，想要调用子类特有的方法，必须做向下转型。

abstract class **Animal** {

    abstract void **eat**();

}

class **Cat** extends **Animal** {

    public void **eat**() {

        System.out.**println**("吃鱼");

    }

    public void **catchMouse**() {

        System.out.**println**("抓老鼠");

    }

}

class **Dog** extends **Animal** {

    public void **eat**() {

        System.out.**println**("吃骨头");

    }

    public void **watchHouse**() {

        System.out.**println**("看家");

    }

}

public class **Test** {

    public static void **main**(String[] args) {

*// 向上转型*

        Animal a = new **Cat**();

        a.**eat**();  *// 调用的是 Cat 的 eat*

*// 向下转型*

        Cat c = (Cat)a;

        c.**catchMouse**();  *// 调用的是 Cat 的 catchMouse*

    }

}

* **类转型异常与instanceof**

转型的过程中，一不小心就会遇到这样的问题，请看如下代码：

public class **Test** {

    public static void **main**(String[] args) {

*// 向上转型*

        Animal a = new **Cat**();

        a.**eat**();               *// 调用的是 Cat 的 eat*

*// 向下转型*

        Dog d = (Dog)a;

        d.**watchHouse**();        *// 调用的是 Dog 的 watchHouse 【运行报错】*

    }

}

这段代码可以通过编译，但是运行时，却报出了 ClassCastException ，类型转换异常！这是因为，明明创建了Cat类型对象，运行时，当然不能转换成Dog对象的。这两个类型并没有任何继承关系，不符合类型转换的定义。

为了避免ClassCastException的发生，Java提供了 **instanceof** 关键字，给引用变量做类型的校验，所以，转换前，我们最好先做一个判断，代码如下：

public class **Test** {

    public static void **main**(String[] args) {

*// 向上转型*

        Animal a = new **Cat**();

        a.**eat**();               *// 调用的是 Cat 的 eat*

*// 向下转型*

        if (a instanceof Cat){

            Cat c = (Cat)a;

            c.**catchMouse**();        *// 调用的是 Cat 的 catchMouse*

        } else if (a instanceof Dog){

            Dog d = (Dog)a;

            d.**watchHouse**();       *// 调用的是 Dog 的 watchHouse*

        }

    }

}

#### 5. 一个多态的综合案例

interface **USB** {

    void **open**();*// 开启功能*

    void **close**();*// 关闭功能*

}

class **Mouse** implements **USB** {

    public void **open**() {

        System.out.**println**("鼠标开启，红灯闪一闪");

    }

    public void **close**() {

        System.out.**println**("鼠标关闭，红灯熄灭");

    }

    public void **click**(){

        System.out.**println**("鼠标单击");

    }

}

class **KeyBoard** implements **USB** {

    public void **open**() {

        System.out.**println**("键盘开启，绿灯闪一闪");

    }

    public void **close**() {

        System.out.**println**("键盘关闭，绿灯熄灭");

    }

    public void **type**(){

        System.out.**println**("键盘打字");

    }

}

class **Laptop** {

*// 笔记本开启运行功能*

    public void **run**() {

        System.out.**println**("笔记本运行");

    }

*// 笔记本使用usb设备，这时当笔记本对象调用这个功能时，必须给其传递一个符合USB规则的USB设备*

    public void **useUSB**(USB usb) {

*// 判断是否有USB设备*

        if (usb != null) {

            usb.**open**();

*// 类型转换,调用特有方法*

            if(usb instanceof Mouse){

                Mouse m = （Mouse）usb；

                    m.**click**();

            }else if (usb instanceof KeyBoard){

                KeyBoard kb = (KeyBoard)usb;

                kb.**type**();

            }

            usb.**close**();

        }

    }

    public void **shutDown**() {

        System.out.**println**("笔记本关闭");

    }

}

public class **Test** {

    public static void **main**(String[] args) {

*// 创建笔记本实体对象*

        Laptop lt = new **Laptop**();

*// 笔记本开启*

        lt.**run**();

*// 创建鼠标实体对象*

        Usb u = new **Mouse**();

*// 笔记本使用鼠标*

        lt.**useUSB**(u);

*// 创建键盘实体对象*

        KeyBoard kb = new **KeyBoard**();

*// 笔记本使用键盘*

        lt.**useUSB**(kb);

*// 笔记本关闭*

        lt.**shutDown**();

    }

}

### 第五节 面向对象的语法补充

#### 1. final关键字

**final**表示不可改变。可以用于修饰类、方法和变量。

1) 修饰类：被修饰的类，不能被继承。

final class **类名** {

}

2) 修饰方法：被修饰的方法，不能被重写。

修饰符 final 返回值类型 方法名(参数列表){

*//方法体*

}

3) 修饰变量：被修饰的变量，不能被重新赋值。

* **局部变量-基本类型**

基本类型的局部变量，被final修饰后，只能赋值一次，不能再更改。代码如下：

public class **FinalDemo1** {

    public static void **main**(String[] args) {

*// 声明变量，使用final修饰*

        final int a;

*// 第一次赋值*

        a = 10;

*// 第二次赋值*

        a = 20; *// 报错,不可重新赋值*

*// 声明变量，直接赋值，使用final修饰*

        final int b = 10;

*// 第二次赋值*

        b = 20; *// 报错,不可重新赋值*

    }

}

在这里有一个坑，看下面两段代码：

final int c = 0;

for (int i = 0; i < 10; i++) {

    c = i;

    System.out.**println**(c);

}

for (int i = 0; i < 10; i++) {

    final int c = i;

    System.out.**println**(c);

}

根据 final 的定义，写法1报错！写法2，为什么通过编译呢？因为每次循环，都是一次新的变量c。这也是大家需要注意的地方。

* **局部变量-引用类型**

引用类型的局部变量，被final修饰后，只能指向一个对象，地址不能再更改。但是不影响对象内部的成员变量值的修改，代码如下：

public class **FinalDemo2** {

    public static void **main**(String[] args) {

*// 创建 User 对象*

        final   User u = new **User**();

*// 创建 另一个 User对象*

        u = new **User**(); *// 报错，指向了新的对象，地址值改变。*

*// 调用setName方法*

        u.**setName**("张三"); *// 可以修改*

    }

}

* **成员变量**

成员变量涉及到初始化的问题，初始化方式有两种，只能二选一：

显示初始化：

public class **User** {

    final String USERNAME = "张三";

    private int age;

}

构造方法初始化：

public class **User** {

    final String USERNAME ;

    private int age;

    public **User**(String username, int age) {

        this.USERNAME = username;

        this.age = age;

    }

}

tips：被final修饰的常量名称，一般都有书写规范，所有字母都大写。

#### 2. 权限修饰符

在Java中提供了四种访问权限，使用不同的访问权限修饰符修饰时，被修饰的内容会有不同的访问权限。



#### 3. 内部类

如果将一个类A定义在另一个类B里面，里面的那个类A就称为内部类，B则称为外部类。在描述事物时，若一个事物内部还包含其他事物，就可以使用内部类这种结构。比如，汽车类 Car 中包含发动机类 Engine ，这时， Engine 就可以使用内部类来描述，定义在成员位置。代码示例：

class **Car** { *//外部类*

    class **Engine** { *//内部类*

    }

}

* **访问特点**

内部类可以直接访问外部类的成员，包括私有成员。外部类要访问内部类的成员，必须要建立内部类的对象。创建内部类对象的格式如下：

外部类名.内部类名 对象名 = new 外部类型().new 内部类型()；

一个成员内部类的使用案例;

public class **Person** {

    private  boolean live = true;

    class **Heart** {

        public void **jump**() {

*// 直接访问外部类成员*

            if (live) {

                System.out.**println**("心脏在跳动");

            } else {

                System.out.**println**("心脏不跳了");

            }

        }

    }

    public boolean **isLive**() {

        return live;

    }

    public void **setLive**(boolean live) {

        this.live = live;

    }

}

public class **InnerDemo** {

    public static void **main**(String[] args) {

*// 创建外部类对象*

        Person p  = new **Person**();

*// 创建内部类对象*

        Heart heart = p.new **Heart**();

*// 调用内部类方法*

        heart.**jump**();

*// 调用外部类方法*

        p.**setLive**(false);

*// 调用内部类方法*

        heart.**jump**();

    }

}

*/\**

*输出结果:*

*心脏在跳动*

*心脏不跳了*

*\*/*

tips：内部类仍然是一个独立的类，在编译之后会内部类会被编译成独立的 .class文件，但是前面冠以外部类的类名和$符号 。比如，Person$Heart.class

* **匿名内部类**

**匿名内部类**是内部类的简化写法。它的本质是一个 带具体实现的 父类或者父接口的 匿名的 子类对象。开发中，最常用到的内部类就是匿名内部类了。以接口举例，当你使用一个接口时，似乎得做如下几步操作，

1). 定义子类

2). 重写接口中的方法

3). 创建子类对象

4). 调用重写后的方法

我们的目的，最终只是为了调用方法，那么能不能简化一下，把以上四步合成一步呢？匿名内部类就是做这样的快捷方式。

使用匿名内部类有个前提，**必须先继承一个父类或者实现一个父接口**。

new 父类名或者接口名(){

*// 方法重写*

    @Override

    public void **method**() {

*// 执行语句*

    }

};

* **使用匿名内部类**

以接口为例，匿名内部类的使用，代码如下：

1) 定义接口：

public abstract class **FlyAble**{

    public abstract void **fly**();

}

2） 创建匿名内部类，并调用：

public class **InnerDemo** {

    public static void **main**(String[] args) {

*/\**

*1.等号右边:是匿名内部类，定义并创建该接口的子类对象*

*2.等号左边:是多态赋值,接口类型引用指向子类对象*

*\*/*

        FlyAble  f = new **FlyAble**(){

            public void **fly**() {

                System.out.**println**("我飞了~~~");

            }

        };

*//调用 fly方法,执行重写后的方法*

        f.**fly**();

    }

}

3) 通常在方法的形式参数是接口或者抽象类时，也可以将匿名内部类作为参数传递。代码如下：

public class **InnerDemo2** {

    public static void **main**(String[] args) {

*/\**

*1.等号右边:定义并创建该接口的子类对象*

*2.等号左边:是多态,接口类型引用指向子类对象*

*\*/*

        FlyAble  f = new **FlyAble**(){

            public void **fly**() {

                System.out.**println**("我飞了~~~");

            }

        };

*// 将f传递给showFly方法中*

**showFly**(f);

    }

    public static void **showFly**(FlyAble f) {

        f.**fly**();

    }

}

4） 以上两步，也可以简化为一步，代码如下：

public class **InnerDemo3** {

    public static void **main**(String[] args) {

*/\**

*创建匿名内部类,直接传递给showFly(FlyAble f)*

*\*/*

**showFly**( new **FlyAble**(){

            public void **fly**() {

                System.out.**println**("我飞了~~~");

            }

        });

    }

    public static void **showFly**(FlyAble f) {

        f.**fly**();

    }

}

#### 4. 引用类型用法总结

* **class 作为成员变量**
* **interface 作为成员变量**
* **interface 作为方法参数和返回值类型**

# 第三篇 Java泛型与容器

第一节

# 第四篇 Java系统编程

# 第五篇 Java网络编程

# 第六篇 Java数据库编程