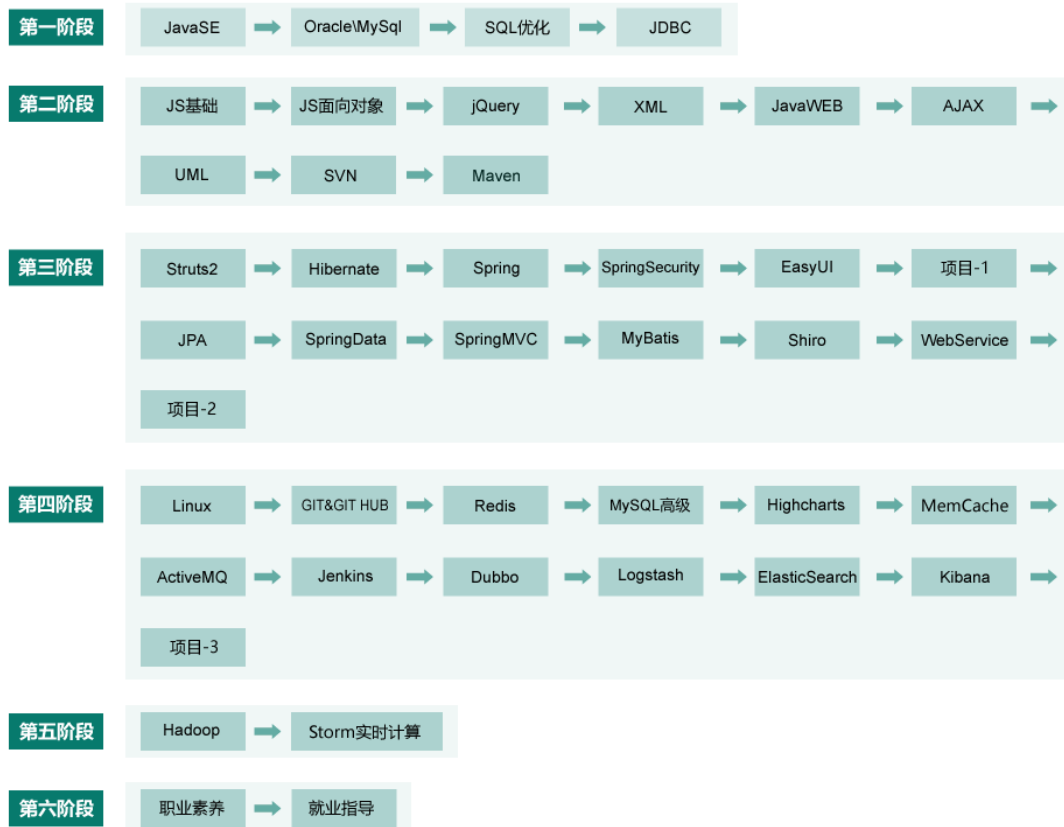


【尚硅谷-IT 精英计划】

JavaSE 学习笔记

视频下载导航 (Java 学习路线图)

JavaEE 学科体系：



Android 学科体系：

第1阶段 Java核心基础	Java基础概述	Java基本语法	面向对象编程	异常处理
	集合框架	泛型	枚举和注解	IO流
	多线程	常用类	反射机制	网络编程
	家庭收支记账软件案例	单机考试管理软件案例	客户信息管理软件案例	开发团队人员调动软件案例
第2阶段 HTML5开发基础	HTML常用标签	HTML5新特性	CSS/CSS3常用语法	JavaScript基本语法

第3阶段 Android基础 (基于Android6.0)	Android开发环境搭建	熟悉使用Android Studio	四大应用组件	界面编程
	数据存储与IO	网络应用	Handler消息机制	事件处理机制
	应用资源	图片三级缓存	Fragment和ViewPager	图形与图像处理
	多媒体应用开发	管理桌面	传感器开发	GPS应用
	Git/SVN			

第4阶段 Android 高级开发	自定义圆形进度条	自定义快速索引	自定义侧滑菜单	自定义滑动删除
	自定义优酷菜单	自定义广告条	自定义下拉刷新	自定义滑动开关
	自定义水波纹	自定义ViewPager	C语言	JNI/NDK开发
	锅炉压力显示案例	应用卸载问卷调查案例	美图处理案例	Android软件的破解技术
	Android系统源码分析	常用第三方框架源码分析	百度地图	

第5阶段 Android 真实项目	图片拾取器	手机卫士	手机影音	百思不得姐
	新闻资讯	硅谷社交	硅谷商城	

第6阶段 Android 重量级框架	Afinal	xUtils3	Volley	OkHttp
	Retrofit	Image-Loader	PhotoView	Picasso
	Fresco	Glide	Gson	FastJson
	EventBus	Otto	GreenDao	SnappyDB
	Butterknife	AndroidAnnotations	TabLayout	ViewPagerIndicator
	SlidingMenu	ActionBar	Toolbar	ActionBarSherlock
	PagerSlidingTabStrip	PinterestLikeAdapterView	NotBoringActionBar	StickyListHeaders
	NineOldAndroids	Android-gif-drawable	Vitamo5.0	PulltoRefresh
	xListView	Expandablelistview	SwipeRefreshLayout	SwipeMenuListView

第7阶段 Android 前沿实用技术	Android主流软件架构搭建	图文混排技术	百分比布局	短信验证技术
	二维码扫描技术	第三方登录 (QQ/微信/微博)	第三方分享	第三方支付 (支付宝/微信等)
	语音识别	消息推送	H5混合开发	APP增量升级
	软件崩溃收集	多渠道打包及软件上线	Android6.0 / 7.0新特性	Android内存优化
	前沿技术分享			

第8阶段 Android+H5 混合开发	React基本语法	React Native开发环境搭建	React Native开发基础	React Native常用组件
	React Native常用API	豆瓣搜索项目		

第9阶段 Android+H5 项目实战	2345影视	风行网	贝瓦儿歌	儿歌多多	蜻蜓FM
	内涵段子	华数TV	电子竞技	穷游	片刻
	时光网	笔趣阁	零食小喵	猫眼电影	
第10阶段 开发经验分享	工作经验分享			面试与就业指导	

-----JavaSE 学习目录-----

第 1 章：Java 语言概述

第 8 章：泛型

第 2 章：基本语法

第 9 章：注解 & 枚举

第 3 章：面向对象编程

第 10 章：IO 流

第 4 章：高级类特性 1

第 11 章：多线程

第 5 章：高级类特性 2

第 12 章：Java 常用类

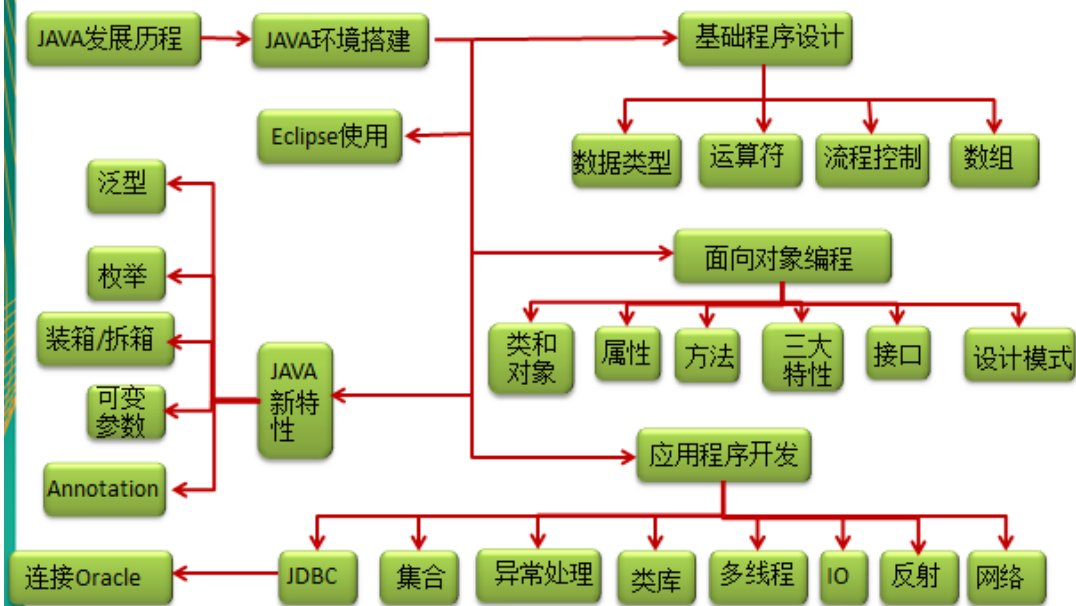
第 6 章：异常处理

第 13 章：Java 反射

第 7 章：Java 集合

第 14 章：网络编程

【基础体系框架】



第 1 章：Java 语言概述

本章内容

- 1.1 基础常识
- 1.2 Java语言概述
- 1.3 Java程序运行机制及运行过程
- 1.4 Java语言的环境搭建
- 1.5 开发体验 — HelloWorld
- 1.6 小结第一个程序
- 1.7 常见问题及解决方法
- 1.8 注释
- 1.9Java API文档

1.1 基础常识

软件：系统软件和应用软件

人机交互方式：图形化界面和命令行方式

常用的 DOS 命令：

dir： 列出当前目录下的文件以及文件夹

md： 创建目录

rd： 删除目录

cd： 进入指定目录

cd..： 退回到上一级目录

cd\： 退回到根目录

del： 删除文件

exit： 退出 dos 命令行

---->学会如何在 DOS 命令下编译并运行 java 源程序（重点）：javac.exe java.exe



1.2 Java 语言概述

➤ 了解语言的分代：第一代：机器语言 第二代：汇编语言 第三代：高级语言（面向过程 & 面向对象）

1.3 Java 程序运行机制及运行过程

Java 语言的特点：①纯面向对象性：类&对象；面向对象的三大特性：封装性、继承性、多态、（抽象）

②健壮性：----->Java 的内存回收机制

③跨平台性：一次编译，到处运行。 ----->JVM

1.4 Java 语言的环境搭建：掌握下载、安装 JDK，并且配置环境变量（重点）

1) JDK 和 JRE 以及 JVM 的关系

2) JDK 的安装

3) 配置 path 环境变量 path：window 执行命令时所需要搜寻的路径。

将 D:\Java\jdk1.7.0_07\bin 复制在 path 环境变量下。

1.5 开发体验 — HelloWorld

```
public class HelloWorld{  
    public static void main(String[] args){  
        System.out.println("HelloWorld!!");  
    }  
}
```

1.6 小结第一个程序

- 1) Java 源文件以“java”为扩展名。源文件的基本组成部分是类（class），如本类中的 HelloWorld 类。
- 2) 一个 Java 源程序中可以存在多个 class 类，但是最多只能有一个类声明为 public。若存在声明为 public 的类，那么这个源程序文件的名字必须以此类的类名来命名
- 3) 程序的入口是 public static void main(String[] args){} 称为主方法。它的写法是固定的。
- 4) Java 方法由一条条语句构成，每个语句以“;”结束。
- 5) Java 语言严格区分大小写
- 6) 大括号都是成对出现的，缺一不可，用于表明类中成员的作用范围。

1.7 常见问题及解决方法

1.8 注释

作用：提高了代码的阅读性；调试程序的重要方法。

三种注释：

当行注释：//注释的内容

多行注释：/*注释的内容*/

文档注释（Java 所特有的，可以为 javadoc 命令所解析）：/** 注释的内容 */

1.9 Java API 文档： API:application programming interface

-----第 2 章：基本语法 -----

本章内容

- 2.1 关键字
- 2.2 标识符
- 2.3 变量
 - 基本数据类型
 - 基本数据类型转换
- 2.4 运算符
- 2.5 程序流程控制
- 2.6 数组

2.1 关键字 & 保留字

被 Java 语言赋予了特殊含义，用做专门用途的字符串（单词）

2.2 标识符

凡是自己可以起名字的地方都叫标识符。

通常有：类名、变量名、方法名。。。包名、接口名、。。。。

规则：（必须按照如下的规则执行，一旦某规则不符合，编译运行时就会出问题）

由 26 个英文字母大小写，0-9，_或 \$ 组成

数字不可以开头。

不可以使用关键字和保留字，但能包含关键字和保留字。

Java 中严格区分大小写，长度无限制。

标识符不能包含空格。

命名的习惯：（如果不遵守，实际上程序编译运行也不会出问题）

包名：多单词组成时所有字母都小写：xxxyyyzzz

类名、接口名：多单词组成时，所有单词的首字母大写：XxxYyyZzz

变量名、方法名：多单词组成时，第一个单词首字母小写，第二个单词开始每个单词首字母大写：xxxYyyZzz

常量名：所有字母都大写。多单词时每个单词用下划线连接：XXX_YYY_ZZZ

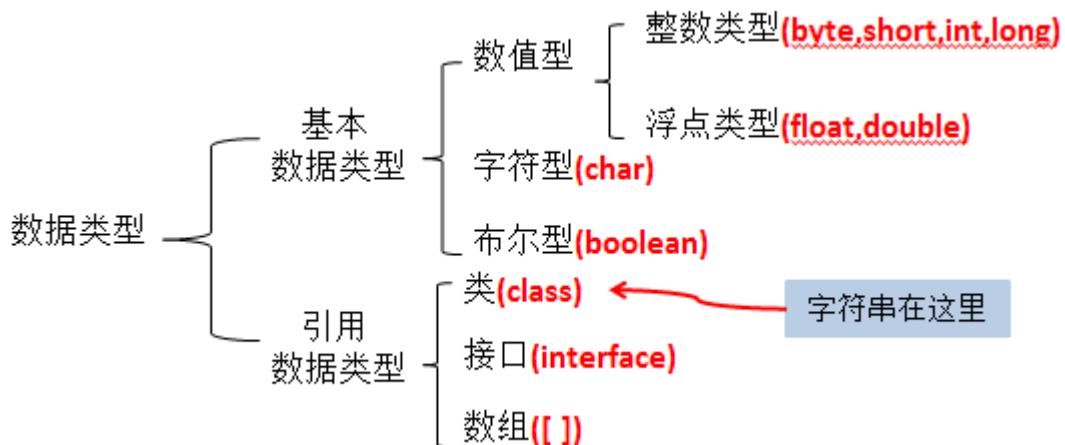
2.3 变量

1.变量的分类

1) 存在的位置的不同：成员变量（属性、Field）(存在于类内部，方法体的外部)和局部变量(方法体的内部、

构造器的内部、代码块的内部、方法体或构造器的形参部分)

2) 数据类型的不同：



基本数据类型（8种）

整型：byte(1个字节)、short、int（默认值类型）、long（后缀是l或L）。

浮点型：float（后缀是f或F）和double（默认值类型）。

布尔型：boolean（只有两个值：true和false，绝对不会取值为null）

字符型：char（1个字符）

引用数据类型：类（比如：String字符串类型）、接口、数组

2.如何声明：

变量的类型 变量的名字 初始化值（显式的初始化、默认初始化）

```
int i;
```

```
i = j + 12;
```

```
boolean b = false;
```

```
String str = "atguigu";
```

```
Customer c = new Customer();
```

3.变量必须先声明后使用

4.使用变量注意:

变量的作用域: 一对{}之间有效

初始化值

5.自动类型转化&强制类型转化(重点)(不包含 boolean 型, 及 String 型)

1) 强制类型转化时, 可能会损失精度。

2.4 进制

十进制

二进制

八进制

十六进制

1.进制之间的转化:

会将-128 到 127 范围内的十进制转化为二进制 & 能将一个二进制数转换为十进制数(延伸: 十进制、二进制、八进制、十六进制)

2.对于几类基本类型的数据的某个具体值, 能够用二进制来表示。同时, 如果给出一个数据的二进制, 要能够会转化为十进制!

正数: 原码、反码、补码三码合一。

负数: 原码、反码、补码的关系。负数在计算机底层是以补码的形式存储的。

2.5 运算符

算术运算符

赋值运算符

比较运算符(关系运算符)

逻辑运算符

“&”和“&&”的区别：&：当左端为 false 时，右端继续执行

&&：当左端为 false 时，右端不再执行

“|”和“||”的区别：|：当左端为 true 时，右端继续执行

||：当左端为 true 时，右端不再执行

位运算符

<< >> >>> & | ^ ~

三元运算符

➤ (条件表达式)?表达式1: 表达式2;

为true, 运算后的结果是表达式1;
为false, 运算后的结果是表达式2;

➤ 表达式1和表达式2为同种类型

1.三元运算符与 if-else 语句的联系

1) 三元运算符可以简化 if-else 语句

2) 三元运算符一定要返回一个结果。结果的类型取决于表达式 1 和 2 的类型。(表达式 1 和 2 是同种类型的)

3) if-else 语句的代码块中可以有多条语句。

2.6 流程控制

条件判断:

```
if(表达式 1){  
    //执行语句  
}else if(表达式 2){  
    //执行语句  
}else{  
    //执行语句  
}
```

- 1.一旦执行条件判断语句，有且仅有一个选择“路径”里的代码块被执行。
- 2.如果多条表达式间是“互斥”关系，彼此是可以调换顺序。

如果多条表达式间存在“包含”关系，一定要将范围小的表达式写在范围大的表达式对应的语句的上面。

- 3.选择判断语句可以“嵌套”
- 4.若执行语句只有一句，那么对应的一对{}可以省略。

选择结构：switch-case

1.结构

```
switch(表达式){  
  case 常量1:  
    语句1;  
    break;  
  case 常量2:  
    语句2;  
    break;  
  ... ..  
  case 常量N:  
    语句N;  
    break;  
  default:  
    语句;  
    break;  
}
```

- 2.表达式可以存放的数据类型：int byte short char String 枚举
- 3.表达式存放是数值或者说是离散的点所代表的常量，绝对不是取值范围。
- 3.default 是可选的。default 的位置也不是固定的，当没有匹配的 case 时，执行 default
- 4.break 在 switch-case 中的使用。当执行到某条 case 语句后，使用 break 可以跳出当前的 switch 语句。

如果此 case 中没有 break，那么，程序将依次执行下面的 case 语句，直至程序结束或者遇到 break。

【switch-case 与 if-else if-else 的联系】

1.表达式是一些离散的点，并且取值范围不是很大，要求是 int byte short char String 枚举类型之一。

建议使用 switch-case。执行效率更高

2.如果表达式表示的是范围（或区间）、取值范围很大，建议使用 if-else if-else

循环结构：①初始化条件②循环条件部分③循环体部分④迭代部分

1.for 循环

```
for(①;②;④){  
    ③  
}
```

执行顺序：①-②-③-④-②-③-④-...-②-③-④-②截止

//死循环：

```
for(;;){  
    //要循环执行的代码。  
}
```

2.while 循环

```
①  
while(②){  
    ③  
    ④  
}
```

//死循环：

```
while(1>0){  
    //要循环执行的代码  
}
```

注意：for 循环和 while 循环之间可以相互转化！

3.do-while 循环

```
①
```

```
do{  
    ③  
    ④  
}while(②);  
//程序至少执行一次!
```

掌握：1.会使用 for 循环和 while 循环
2.能够实现 for 循环和 while 循环的相互转化。

循环可以相互嵌套

例题 1.九九乘法表；

例题 2.输入 1-1000 之间的所有质数

2.7 关键字：break & continue

break

- 1.使用范围：循环结构或 switch-case 结构中。
- 2.在循环结构中：一旦执行到 break，代表结束当前循环。

continue

- 1.使用范围：循环结构
- 2.在循环结构中：一旦执行到 continue,代表结束当次循环。

相同点：如果 break 或 continue 后面还有代码，那么这些代码将不会被执行。所以当有代码，编译会出错！

如何使用标签，实现结束指定“层次”的循环。（理解）

例题：

```
class Test{  
    public static void main(String[] args){  
        for(int i = 1 ;i <= 100;i++){
```

```

        if(i % 10 == 0){
            break;
            //continue;
            //下面不能写入任何代码
        }
        System.out.println(i);
    }
}
}

```

2.8 数组

1.如何创建一维数组

1) 数组的声明: `int[] i; String[] str; Animal[] animal;`

2)数组的初始化:

①动态初始化: `i = new int[4];` //i[0]开始, 至 i[3]结束。默认初始化值都为 0;

`i[0] = 12;i[1] = 34;....`

②静态初始化: `str = new String[]{"北京","尚硅谷","java0715 班"};`

//也是先有默认初始化赋值, 然后显示的初始化, 替换原来的默认初始化值

//对于引用数据类型来说, 默认初始化值为 null。

//对于基本数据类型来说: `byte、short、int : 0` `long : 0L` `float : 0.0F` `double :`

`0.0` `char : '\u0000'` `boolean : false` 引用类型: `null`

2.数组元素的下角标从 0 开始。

3. `float[] f = new float[]{1.2F,3.4F,5.6F};`

`double[] d = new double[4]; d[1] = 3.4;.....`

4.数组的长度: `.length;`

5.数组的遍历: (习惯使用 for 循环)

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

```
    System.out.print(f[i] + "\t");
```

```
}
```

6.常见的异常:

1) `ArrayIndexOutOfBoundsException` :数组下标越界异常 下标从 0 开始, 到 `length-1` 结束。如果下角标的取值不在此范围内, 将报此异常

2) 空指针异常(`NullPointerException`)

2.二维数组

1)

①动态初始化:

```
---> int[][] i = new int[3][];    i[0] = new int[2];    i[1] = new int[3];    i[2] = new int[4];  
--->String[][] str = new String[2][4];
```

② 静态初始化:

```
int[][] i = new int[][]{{1,2,3},{3,4},{0}};
```

2)遍历 (补充完整)

7. 数组常用的算法: 最大值、最小值、和、平均数、排序 (涉及数据结构中各种排序算法)、复制、反转。

8. `Arrays`: 操作数组的工具类

`Array.sort(数组类型形参);` 可以对形参部分的数组进行排序, 默认是从小到大的顺序。

学习内容

- 3.1 面向对象与面向过程
- 3.2 java语言的基本元素：类和对象
- 3.3 类的成员之一：属性
- 3.4 类的成员之二：方法
- 3.5 对象的创建和使用
- 3.6 再谈方法
- 3.7 面向对象特征之一：封装和隐藏
- 3.8 类的成员之三：构造器（构造方法）
- 3.9 几个关键字：this、package、import

3.1 面向过程与面向对象

- 1.面向过程关注于功能和行为
面向对象关注于功能和行为所属的对象。
- 2.Java 语言，作为面向对象的语言，更多的关注于类的设计！
- 3.面向对象两大元素：类和对象
三大特性：封装、继承、多态、（抽象）

3.2 类

- 1.java 的源程序是由一个一个的类构成的。

源文件名【类名 3.java】

```
class 类名 1{  
    属性 1  
    属性 2  
    ...  
    方法 1(){  
    }  
    方法 2(){  
    }  
    .....  
}
```

```
class 类名 2{
```

```
}
```

```
public class 类名 3{
```

```
}
```

2.类的组成部分：1) 属性 2) 方法 3) 构造器 4) 代码块 5) 内部类

1) 属性 (Field、成员变量、字段)：定义在类内部，方法外部的变量。

①格式：修饰符(public private 缺省 protected) 数据类型 名字 = 初始化值

②属性的初始化值：如果不显式初始化的话，系统会根据属性的数据类型，隐式初始化。

当你显式的给属性赋值，那么属性经过默认初始化-->显式初始化这样的步骤。

③相对于成员变量，有局部变量

局部变量：在方法体内部（或在代码块内部或方法的形参部分）定义的变量。

格式： 数据类型 名字 = 初始化值

初始化值：必须要显式的初始化。系统不会提供默认初始化值。

成员变量和局部变量的区别与联系：

* 相同点：1) 变量，在声明的时候，都需要指定数据类型和变量名。

* 2) 都有生命周期。

* 不同点：1) 声明的位置不同

* 2) 在内存中的加载不同：成员变量随着 new Person()的加载，而加载到堆空间中

* 局部变量是加载在栈空间。

* 3) 初始化值：成员变量：有默认初始化值，当然也可以显式的初始化

* 局部变量：除形参部分的局部变量外，必须显式的初始化

* 4) 访问修饰符：成员变量：需要有访问修饰符.访问权限从大到小有：public protected 缺省状态 private

* 局部变量：不需要有访问修饰符。实际上它的访问权限由其所在的方法的权限所反映。

2) 方法(method、函数、成员方法)

①格式：（写一个方法，从如下声明的 4 部分去考虑）

修饰符 返回值类型 方法名(形参 1 类型 形参 1 名字,形参 2 类型 形参 2 的名字,....){

//方法体

}

②说明：1.在方法内部，可以调用当前类的属性。（有一个例外：在 `static` 声明的方法里，不能调用非 `static` 的属性。）

2.方法内部，可以声明局部变量

3.方法内部可以调用其他方法，但是不能定义方法。

* 方法的返回值类型：有返回值的 & 无返回值的

* 有返回值的：在方法声明时，指定返回值的类型。在方法体的里面需要 `return` + 返回值类型的“实体”；

* 无返回值的：在方法声明的返回值类型位置写上 `void`.此时不需要返回值，即无 `return`.

*

* 记忆：`void` 和 `return` 像是一对冤家，你出现，我就躲起来不出现。

3.3 对象的创建

1. 创建对象的格式：

类名 类名的一个引用 = `new` 类对应的构造器；

比如：`String str = new String("atguigu");`

`Person p = new Person();`

创建类的多个对象，每个对象都在堆空间有独立的一块区域。对 `a` 对象的区域进行的操作，并不影响 `b` 对象或其它对象区域的内容。

2.匿名对象的创建

不定义对象的句柄，而直接调用这个方法。这样的对象叫做匿名对象。

如：`new Person().shout();`

使用情况：

如果对一个对象只需要进行一次方法调用，那么就可以使用匿名对象。

我们经常将匿名对象作为实参传递给一个方法调用。

3.4 方法的重载

方法的重载（`overload`）：

1.满足的条件：“两同一不同” 同一个类中，同一个方法名，参数列表不同（参数类型，参数个数的不同）

注意：至于方法的权限修饰符和返回值类型并不影响方法之间是否构成重载
参数名不作为是否构成方法重载的条件。

//返回两个整数的和

```
int add(int x,int y){return x+y;}
```

//返回三个整数的和

```
int add(int x,int y,int z){return x+y+z;}
```

//返回两个小数的和

```
double add(double x,double y){return x+y;}
```

3.5 形参的参数传递

swap(int i ,int j)与 swap(int[] arr,int i ,int j)的区别

m,n ar 0x1234

Java 的实参值如何传入方法呢?

Java 里方法的参数传递方式只有一种：值传递。即将实际参数值的副本（复制品）传入方法内，而参数本身不受影响。

- 1.如果是基本数据类型：就将基本数据类型的值赋给方法的形参部分。
- 2.如果是引用数据类型：就将引用变量 a 的地址值赋给方法的形参部分 b, 这样，方法的形参部分 b 和引用的变量 a 就指向同一份堆空间的内存区域。

3.6 可变形参

- * 测试可变个数的形参。
 - * 1.是指方法的参数部分的指定类型的参数个数是可变的。
 - * 2.个数可以从 0 个开始，到无穷多个
 - * 3.格式: method(数据类型 ... 数据的引用名){}
 - * 4.可变参数方法的使用与方法参数部分使用数组是一致的。并且这个两种参数对应的方法名必须不同。
 - * 如: method2(String[] str){}与 method2(String ... str){}不能同时出现在一个类中。
 - * 5.定义可变参数方法时，要将可变参数写在参数部分的最后。---->一个方法中，最多有一个可变个数的形参!!
 - * 6.method1(String ... str){}方法与 method1(String str){}构成重载

```
public void method1(String str){
    System.out.println(str);
}
```

```
public void method2(String[] str){
    for(int i = 0;i < str.length;i++){
        System.out.print(str[i] + "\t");
    }
}
```

```
public void method1(String ... str){
    for(int i = 0;i < str.length;i++){
        System.out.print(str[i] + "\t");
    }
}
```

3.7 递归方法

方法递归：一个方法体内调用它自身。

方法递归包含了一种隐式的循环，它会重复执行某段代码，但这种重复执行无须循环控制。

例题：已知有一个数列： $f(0) = 1, f(1) = 4, f(n+2) = 2 * f(n+1) + f(n)$, 其中 n 是大于 0 的整数，求 $f(10)$ 的值。

递归一定要向已知方向递归，否则这种递归就变成了无穷递归，类似于死循环。

3.8 构造器

1.构造器(constructor)的作用：1) 创建类的对象 2) 通过构造器的形参赋值，初始化对象的属性

2.如果声明一个类的构造器

1) 构造器与类同名，有修饰符，有形参，没有返回值也无 void。

```
class A{
    public A(){
        //构造器内部代码
    }
}
```

2) 类内部可以声明多个重载的构造器。

3.构造器的说明：

- 1) Java 语言中，每个类都至少有一个构造器
- 2) 默认构造器的修饰符与所属类的修饰符一致
- 3) 一旦显式定义了构造器，则系统不再提供默认构造器
- 4) 一个类可以创建多个重载的构造器
- 5) 父类的构造器不可被子类继承

3.9 封装与隐藏

通过设置类的属性为 private,使用 public 的方法调用和修改类中 private 的属性。它是封装和隐藏的一种表现形式。

```
public class Animal{
    private int legs;//将属性 legs 定义为 private，只能被 Animal 类内部访问
    public void setLegs(int i){ //在这里定义方法 eat() 和 move()
        if (i != 0 && i != 2 && i != 4){
            System.out.println("Wrong number of legs!");
            return;
        }
        legs=i;
    }
    public int getLegs(){
        return legs;
    }
}
```

3.10 四种权限修饰符

public protected 缺省 private

修饰符	类内部	同一个包	子类	任何地方
private	Yes			
(缺省)	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

对于class的权限修饰只可以用public和default(缺省)。

- public类可以在任意地方被访问。
- default类只可以被同一个包内部的类访问。

3.11 this

1. this 可以用来修饰属性、方法和构造器

1) this 修饰属性和方法表示调用当前对象（或正在初始化的对象）的属性和方法。

```
public void display(){ //调用 display()方法的对象，即为当前对象
    this.name = "Lily";
    this.setAge(12);
}
```

```
public Person(String name,int age){ //此时 this 表示：正在初始化的对象
    this.name = name;
    this.age = age;
}
```

2) 在类内部，构造器内，使用 this（参数列表）的形式，调用本类重载的构造器

要求：1) this（参数列表）一定要写在构造器的首行！

2) 在类内部的构造器中，至少有一个构造器内部是没有 this(参数列表)的。

3.12 package & import

package :包。

写在源文件的第一行，声明源文件的“位置”。

```
package com.atguigu.java;
```

import 引入

如: `import java.util.Scanner;` 表示导入 `java.util` 包下的某一个类, 即 `Scanner` 类
`import java.sql.*;`表示导入 `java.sql` 包下的所有的类

然后, 可以在程序中, 直接使用。即: `Scanner s = new Scanner(System.in);`

1)如果需要导入 `java.lang` 的话, 可以省略不写。

2) 如果不显式的 `import` 的话, 也可以。 `java.util.Scanner s = new java.util.Scanner(System.in);`

3) 像 `Date` 这样, 既存在于 `java.util` 包下, 又存在于 `java.sql` 包下, 如果在源程序中同时出现了两个 `Date` 类, 那么只能是使用 2) 表示的方式。

`import static` 在 `JDK1.5` 引入的。表示导入某个类下的某个静态方法、属性或全部的静态方法或属性

```
import static java.lang.Math.PI;
import static java.lang.System.*;
```

第 4 章：高级类特性 1

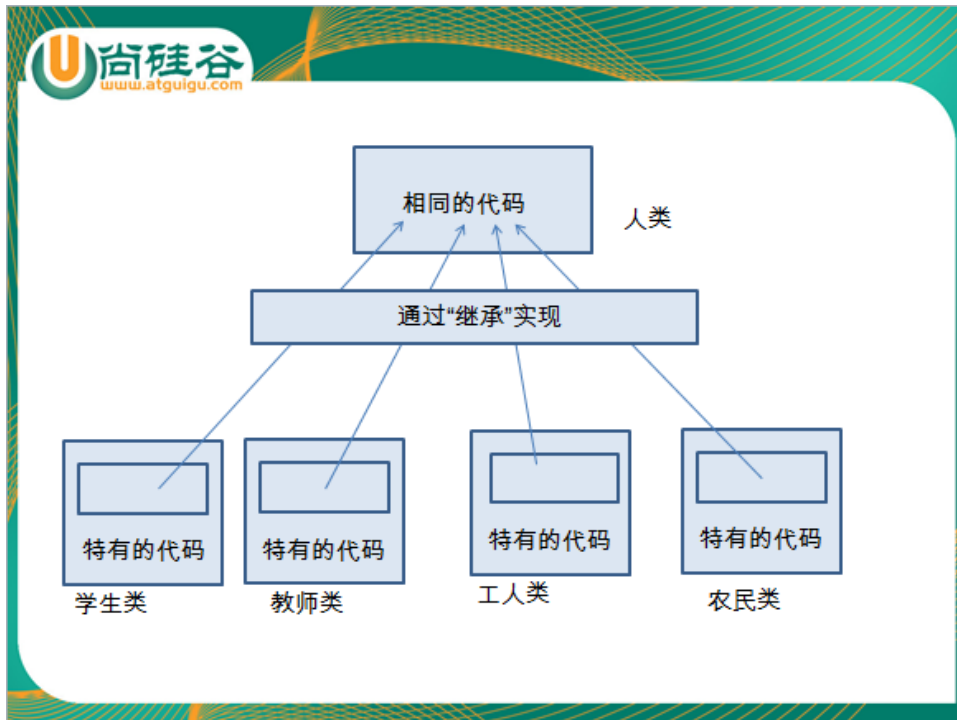


本章内容

- 4.1 面向对象特征之二：继承
- 4.2 方法的重写(override)
- 4.3 四种访问权限修饰符
- 4.4 关键字super
- 4.5 子类对象实例化过程
- 4.6 面向对象特征之三：多态
- 4.7 Object类、包装类

4.1 继承

1.继承的想法



2.继承的格式 `class A extends B{`
`}`

其中，A 叫做子类，叫做父类

1) 通过继承，子类 A 就获得了父类 B 的属性和方法。在父类权限修饰符允许的情况下，子类可以直接调用父类含有的属性和方法。

2) 子类不能继承父类的构造器。但是子类可以通过 `super`（参数列表）的形式调用父类的构造器

3) java 中只能单继承

```
class C extends B{
class D extends A{
```

4) 在 java 语言中，所有类的根父类是 `java.lang.Object` 类。

4.2 方法的重写

1.重写的前提：一定要有继承；

2.重写的要求：1) 返回值类型、方法名、形参列表（形参个数，形参类型）跟父类相同

2) 访问权限：子类重写方法的访问权限不能小于父类被重写方法的访问权限

3) 关于异常：子类抛出的异常不能大于父类被重写的方法抛出的异常

4) 关于 `static`：重写和被重写的方法，必须同时为 `static` 或者同时为非 `static` 修饰的。

【重载（overload）和重写（override）的区别】

1. 答出重载与重写对方法声明的要求。
2. 重载：在同一个类中。
重写：是在子类中，“重写”的父类的方法。

4.3 super 关键字

super：对父类的方法、属性或构造器的引用。

```
class Person{
    String name = "1";
}

class Student extends Person{
    String name = "2";
    public void display(){
        System.out.println(super.name);
        System.out.println(this.name);
    }

    public static void main(String[] args){
        Student s = new Student();
        s.display();
    }
}
```

super 修饰构造器：在子类的构造器中，可以通过显示的调用 super(参数列表)的形式，调用指定父类的构造器。
如果不显式的调用的话，子类的构造器也会默认去调用父类的 super()的构造器!!

4.4 子类对象实例化的全过程

```
class Person{
    String name = "1";
}

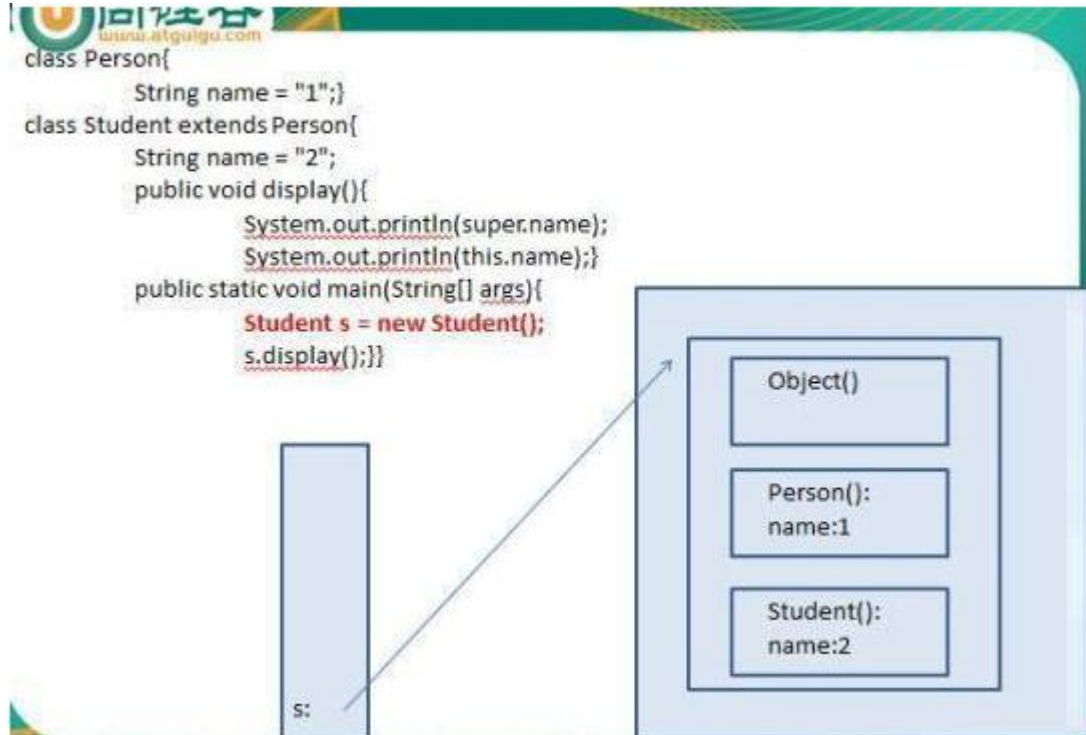
class Student extends Person{
    String name = "2";
    public void display(){
        System.out.println(super.name);
        System.out.println(this.name);
    }

    public static void main(String[] args){
        Student s = new Student();
    }
}
```

```

        s.display();
    }
}

```



4.5 多态性

1. Java 中多态的表现形式:

- 1) 方法的重载(overload)和重写(override)。
 - 2) 对象的多态性 —— 可以直接应用在抽象类和接口上
2. 对象的多态性使用的前提: 1) 有继承关系 2) 子类要重写父类的方法

3. Java 程序分为编译时状态和运行时状态。

```

class Person{
    public void method1(){
        //方法体
    }
}

class Student extends Person{

    public void method1(){
        //方法体, 重写父类的 method1()方法
    }

    public static void main(String[] args){
        Student st = new Student();

```

Person p = new Student();//对象的多态: 即将子类的对象 new Student() 赋给父类的引用 p。

```
//虚拟方法调用
p.method1();//执行的是 Student 类中的重写的 method1()方法！
```

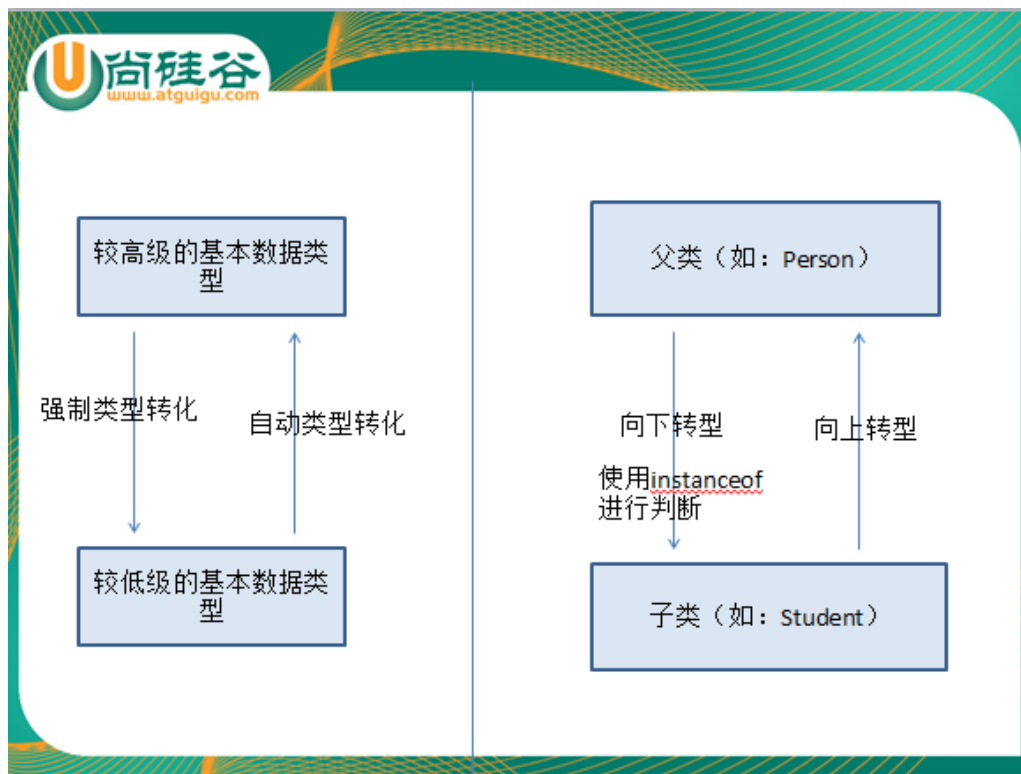
```
//p....是无法调用 Student 类特有的属性和方法的！编译不通过。
//但是堆空间的对象中，是含有 Student 类所特有的属性和方法的！
```

//如何实现 p 可以调用 Student 类所特有的属性和方法呢？造型（或向下转型、强转）。

//为了保证不报 ClassCastException ， 需要使用 instanceof 关键字进行转型前的判断。

```
}
}
```

```
public class Test{
    public void func(Person p){
        p.method1();
    }
}
```



4.6 Object 类

NO.	方法名称	类型	描述
1	public Object()	构造	构造方法
2	public <u>boolean</u> equals(Object obj)	普通	对象比较
3	public <u>int</u> hashCode()	普通	取得Hash码
4	public String <u>toString</u> ()	普通	对象打印时调用

1. 只有一个构造器 public Object();每次我们创建任何一个类的对象的时候，都会调用到 Object 类的无参的构造器！

2. public boolean equals(Object obj){}

Object 类的原码：

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

2.1) 对于 JDK 提供的 String、包装类、Date 等，已经重写了 Object 类的 equals() 方法。

如：String 类重写的 equals(Object object)方法如下：

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String) anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
}
```

```
}  
return false;  
}
```

2.2) 自定义类的话, 如果需要使用 equals 方法, 那么就必须重写 Object 类的 equals();

//自定义类重写了 Object 类的 equals()方法

```
public boolean equals(Object obj) {  
    if (this == obj)//引用地址是一样的, 指向同一块堆空间的区域  
        return true;  
  
    if (obj instanceof Person) {  
        Person p = (Person) obj;  
        return (this.name.equals(p.name) && this.age == p.age);  
    }  
  
    return false;  
}
```

```
3.public String toString(){  
    }
```

Object 类的源码:

```
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

System.out.println(p); 相当于 System.out.println(p.toString());

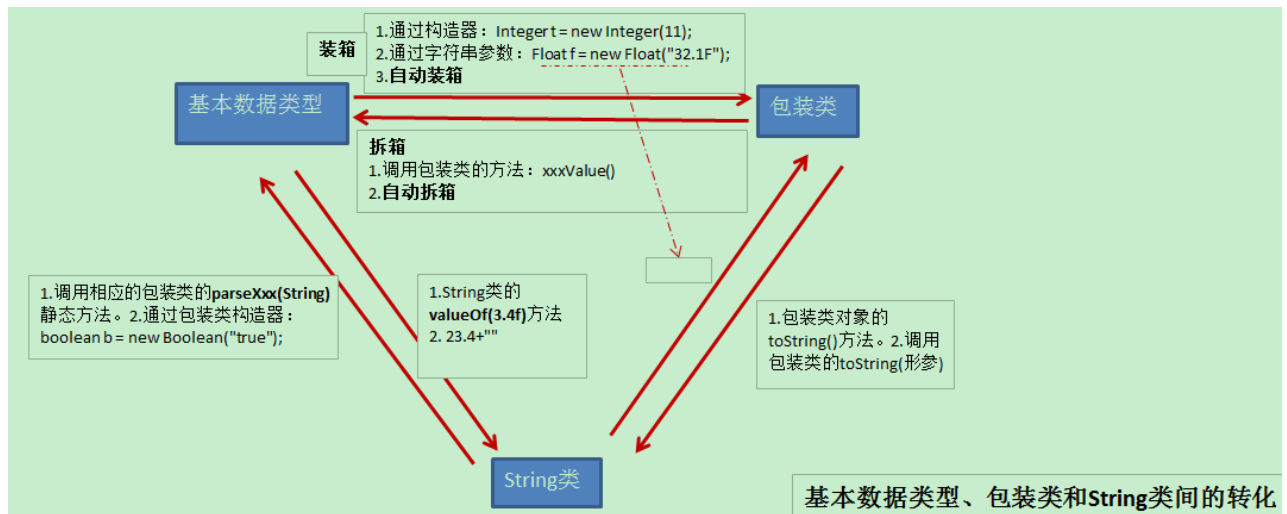
3.1)对于 JDK 提供的 String、包装类、Date 等, 已经重写了 Object 类的 equals()方法。

3.2)自定义类的话, 如果需要使用 toString()方法, 那么就必须重写 Object 类的 toString();

//自定义类重写了 Object 类的 toString()方法

```
public String toString() {  
    return "Person [name=" + name + ", age=" + age + "];"  
}
```

4.7 包装类



//对于基本数据类型和对应的包装类来说，有自动装箱和拆箱

```
Integer i = 23;//自动装箱
Integer j1 = new Integer(34);
Integer j2 = new Integer("34");
Boolean b = new Boolean("true");
int j = j1;
int j3 = j2;//自动拆箱
j1 = null;
boolean boo = b;
System.out.println(boo);//false    注意：不会报异常！
```

```
String str = "123";
//将 String 类型转化成基本数据类型(或包装类) :使用 Xxx 包装类的
parseXxx(String str)方法
```

```
int k = Integer.parseInt(str);
System.out.println(k);
```

```
//基本数据类型(或包装类)转成 String 类
```

```
String str1 = String.valueOf(boo);
System.out.println(str1);
```

```
String str2 = boo + "";
```

```
String str3 = Boolean.toString(boo);
System.out.println(str3);
String str4 = b.toString();
```

5.1 关键字: static 本章内容

- 类变量、类方法
- 单例(Singleton)设计模式

5.2 理解main方法的语法

5.3 类的成员之四: 初始化块

5.4 关键字: final

5.5 抽象类(abstract class)

- 模板方法设计模式(TemplateMethod)

5.6 更彻底的抽象: 接口

- 工厂方法(FactoryMethod)和代理模式(Proxy)

5.7 类的成员之五: 内部类

5.1 static 关键字

1.static :静态的 实现功能: 随着类的加载而加载, 要早于对象的出现

2.static 可以用来修饰: 属性、方法、代码块、内部类

3.static 修饰属性(类属性)

1) 随着类的加载而加载, 而类的消亡而消亡

2) 先于对象创建

3) 可以直接用类来调用

4) 在内存中存在于数据区, 只有一份。不同的对象共同使用同一份 static 的属性

5) 当一个对象对 static 的属性进行修改, 会导致其他对象调用此属性时, 是修改过的属性。

4.static 修饰方法:

1) 随着类的加载而加载, 而类的消亡而消亡

2) 先于对象创建

3) 可以直接用类来调用

4) 在内存中存在于数据区，只有一份。不同的对象共同使用同一份 `static` 的属性

5) 在 `static` 的方法内，只能调用 `static` 的属性或方法。

5.用 `static` 修饰属性或方法的利弊

利：可以直接通过类来调用，不用创建对象

弊：生命周期长，占用内存。

5.2 单例模式

单例模式：要求类只能创建一个对象。

懒汉式 & 饿汉式（存在线程安全问题）

5.3main 方法

```
public static void main(String[] args){}
```

5.4 代码块（或初始化块）

代码块（初始化块）： 用来初始化对象。

1. 静态代码块 和 非静态代码块

*静态代码块：

* 1.代码块内可以有多条语句：输出语句，赋值语句（只能对 `static` 的属性赋值）

* 2.随着类的加载而加载

* 3.静态的代码块只被加载一次

* 4.静态代码块的执行要先于非静态的代码块

* 5.只能调用静态的变量或方法

*

* 非静态的代码块：

* 1.代码块内可以有多条语句：输出语句，赋值语句（对 `static` 的或非 `static` 的属性赋值）

* 2.随着对象的创建才被加载

* 3.每次创建一个对象，非静态代码块都会被执行

* 4.非静态的代码块执行的先后顺序按程序中其出现的先后顺序来执行。

* 5.可以调用静态或非静态的属性和方法。

重难点：创建一个对象，那个对象实例化的全过程。

1) 子类和父类的问题

2) 子类来说，属性的默认/显式初始化，代码块（静态/非静态），构造器。。。

经典的代码：

```
class Root{
    static{
        System.out.println("Root 的静态初始化块");
    }
    {
        System.out.println("Root 的普通初始化块");
    }
    public Root(){
        System.out.println("Root 的无参数的构造器");
    }
}

class Mid extends Root{
    static{
        System.out.println("Mid 的静态初始化块");
    }
    {
        System.out.println("Mid 的普通初始化块");
    }
    public Mid(){
        System.out.println("Mid 的无参数的构造器");
    }
    public Mid(String msg){
        //通过 this 调用同一类中重载的构造器
        this();
    }
}
```

```

        System.out.println("Mid 的带参数构造器，其参数值： "
            + msg);
    }
}
class Leaf extends Mid{
    static{
        System.out.println("Leaf 的静态初始化块");
    }
    {
        System.out.println("Leaf 的普通初始化块");
    }
    public Leaf(){
        //通过 super 调用父类中有一个字符串参数的构造器
        super("atguigu");
        System.out.println("执行 Leaf 的构造器");
    }
}
public class TestLeaf{
    public static void main(String[] args){
        new Leaf();
        new Leaf();
    }
}

```

5.5 final 关键字

* final:表示：最终的，可以用来修饰类、属性、方法

* 1.修饰类：修饰的类不能被继承。

*

* 2.修饰方法：这个方法不能被重写

*

* 3.修饰属性：这个属性值，不能被修改。即，此变量是一个常量。

* “哪里”可以为 final 的属性赋值：1.声明的时候 2.在代码块内部 3.构造器中

*

* `static final` 修饰变量 : 全局常量。

5.6 抽象类

`abstract`:抽象的，可以用来修饰类，修饰方法。

1.使用抽象类的前提：在类可以继承的基础上。

2.抽象类：`abstract class` 类名{} 表明此类不能被实例化。

2.1 抽象类内部可以有构造器！

2.2 抽象类内部不一定有抽象方法。

如：`abstract class Person{`

```
    abstract void walk();
```

```
    abstract void eat();
```

```
}
```

```
class Test{
```

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

```
        Person p = new Person();//不能被实例化
```

```
        Person p1 = new Person(){
```

```
            void walk(){
```

```
                //方法体
```

```
            }
```

```
            void eat(){
```

```
                //方法体
```

```
            }
```

```
        };
```

```
    }
```

2.抽象方法：用 `abstract` 声明，同时，方法没有方法体！

如：`public abstract void walk();`

2.1 抽象方法所属的类一定是抽象类。

2.2 抽象方法只有被重写才有意义。(需要子类继承抽象类, 然后重写抽象方法)

3. `abstract` 不能跟哪些关键字同时使用: `final` `static` `private` 构造器

4. 设计模式: 模板设计模式

5.7 接口

接口: 彻底的抽象类, 不可以被实例化

```
声明: interface MyInterface{  
    //属性(全局常量)  
    public static final double PI =3.14;  
    int E = 2;  
    //方法 (全部是抽象方法)  
    public abstract int getId();  
    void display();  
}
```

1. 格式: `(abstract) class MyClass extends Object implements MyInterface,Comparable{//可以实现多个接口`
`//重写 (所有的/部分的) 抽象方法`
`}`

2. 一个类可以实现多个接口。

3. 类需要重写接口中的所有抽象方法, 方可实例化。如果不的话, 那么此类仍未抽象类

4. 接口与接口间是可以继承的, 而且可以多继承

5. 设计模式: 工厂设计模式、代理模式

5.8 内部类

1. 内部的分类: 成员内部类 (静态的/非静态的) & 局部内部类

```
class Person{  
    String name = "Tom";
```

```

class Bird{
    String name = "小鸟";
    //4 的举例放在此处
}
static clas Dog{

}
}

```

2.成员内部类来说： 1) 创建非静态内部类的实例： 1.1 创建外部类的实例 1.2 通过外部类的对象调用内部类的构造器

```

//1.
Person p = new Person();
//2.
Person.Bird b = p.new Bird();

```

2)创建静态内部类的实例:直接通过外部类的类名来调用内部类的构造器。

```

Person.Dog d = new Person.Dog();

```

3.内部类作为类的一类成员，可以调用外部类的属性和方法。

注意：如果内部类是 `static` 的，那么只能调用外部类的 `static` 的属性和方法。

4.在内部类中调用的问题（尤其是当外部类的属性、内部类的属性、内部类的局部变量名都相同）

举例：

```

public void display(String name){
    System.out.println(name);//黄鹂
    System.out.println(this.name);//小鸟
    System.out.println(Person.this.name);//Tom
}

```

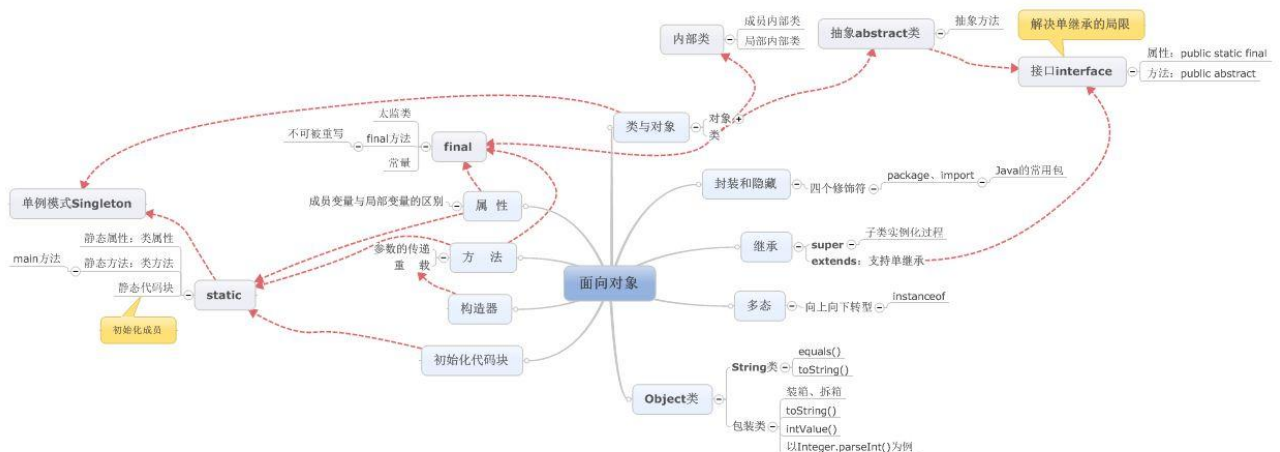
5.局部内部类：在方法里面定义的类。

//与上面定义的 get()方法是一样的。

```
public Comparable get1(){  
    class MyComparable implements Comparable{  
        public int compareTo(Object obj){  
            retrun 0;  
        }  
    }  
  
    return new MyComparable();  
}
```

6.匿名内部类

5.9 面向对象总结



1.面向对象“三条主线”

----->关注与类的创建

1.1 考虑类内部都可以有什么？

属性 方法 构造器 代码块 内部类

1.2 面向对象的特征是什么？

封装和隐藏(四个权限修饰符) 继承 extends 多态

1.3 其他（几个重要的关键字、杂项）

几个重要的关键字：this super static final abstract interface package

import

杂项：Object 类 包装类（包装类、基本数据类型、String 间的转化） main 函数 可变形参 方法的参数传递（难点）

关键点：子类对象实例化的全过程（类的内部哪些部分可以给属性赋值，以及先

后顺序；子父类间的调用)

-----第 6 章：异常处理 -----

1.Exception : RuntimeException(运行时异常)和 CheckedException(编译时异常)

1.1 运行时异常：可以不显式的进行处理。如果出现异常，会将异常的信息显式在控制台上。

几类常见的运行时异常：(4 种)

1.2 编译时异常：编译就不通过。必须要求程序员进行显式的处理！将编译时异常转化为运行时异常。

2.如何处理异常（处理异常的方式）

2.1 try-catch

```
try{
    //有可能出现异常的代码
} catch(Exception1 e1){
    //如何处理的代码；
}catch(Exception1 e2){
    }finally{
    //一定会被执行的代码
}
```

说明：

1) try 中存放的是可能有异常的代码。如果 try 代码中发生异常，那么异常代码之后的代码不会被执行。

try 当中声明的变量，在出了 try 代码块后，将无效

2) catch 语句将异常类型小的放在异常类型大的上面。

在{}中给出异常的处理意见: e.printStackTrace() sysout(e.getMessage);

3)finally:不管是否发生异常，都会被执行: try 中是否有异常; catch 中是否有异常; catch 中是否有 return

4)catch 和 finally 是可选。

5)异常可以嵌套

2.2 throws :如果在方法中抛出了异常，但方法不知道如何处理，那么就通过 throws 的形式，将异常往上抛,交给方法的调用者来处理。

```
举例： public void method() throws Exception{  
        //存在异常  
    }
```

3.异常，既可以是 JDK 提供给我们的异常类，也可以是我们自己定义的异常类。

3.1.1 如果是 JDK 提供给我们的话，那么当出现异常的时候，程序会自动生成一个对应异常类的对象，然后抛出此对象。

3.1.2 如果是自定义的异常，需要在程序中显式的将这个异常对象抛出。如何抛出？使用 throw 关键字

3.2 throw 关键字：定义在方法内部，通过“throw 自定义异常类的对象”格式，抛给方法。

3.3 如何自定义异常类？①自定义异常类继承现有的异常类（一般情况下，继承 RuntimeException）②写几个重载的构造器

③static final long serialVersionUID = 948L;

【典型例题】

```
class EcDef extends RuntimeException{  
    static final long serialVersionUID = 9L;  
    public EcDef(String name){  
        super(name);  
    }  
}  
  
public class EcmDef {  
    public static void main(String[] args) {  
        try{  
            int i = Integer.parseInt(args[0]);  
            int j = Integer.parseInt(args[1]);  
            ecm(i,j);  
        }  
    }  
}
```



```

    }catch(NumberFormatException e){
        System.out.println("数据类型转化的异常");
    }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("数组下标越界的异常");
    }catch(ArithmeticException e){
        System.out.println("算术异常： / by zero");
    }catch(EcDef e){
        System.out.println(e.getMessage());
    }finally{
        System.out.println("程序执行完毕！ ");
    }
}

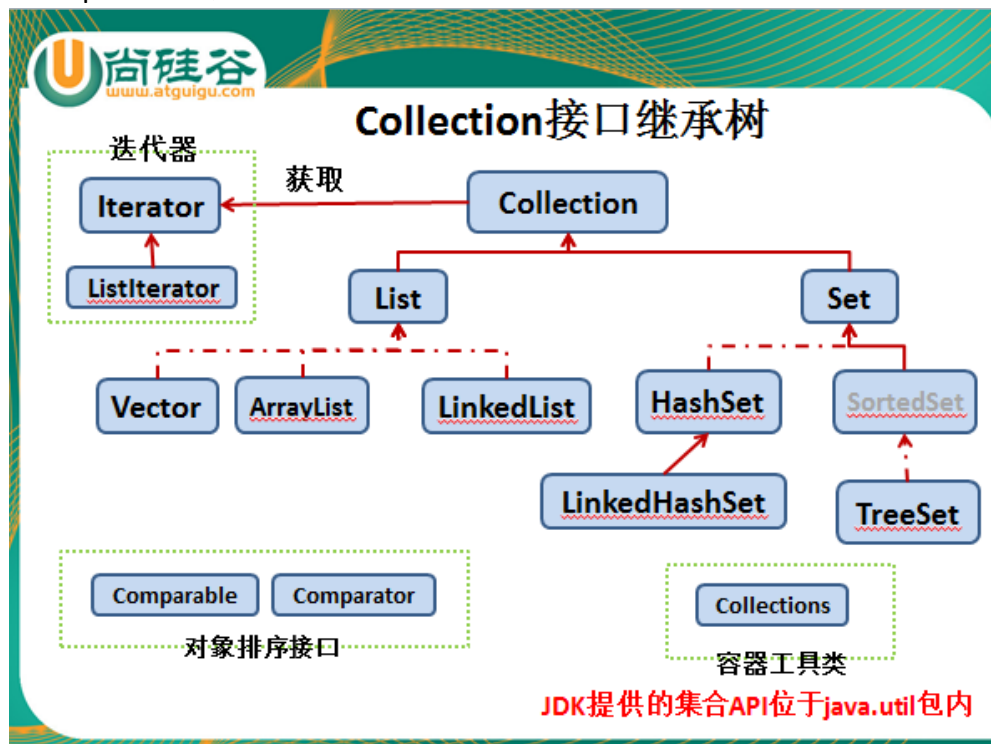
public static void ecm(int m,int n) throws RuntimeException{
    if(m < 0 || n < 0){
        throw new EcDef("不能输入负数！ ");
    }else
        System.out.println("两数相除的结果为： " + m/n);
}
}

```

本章内容

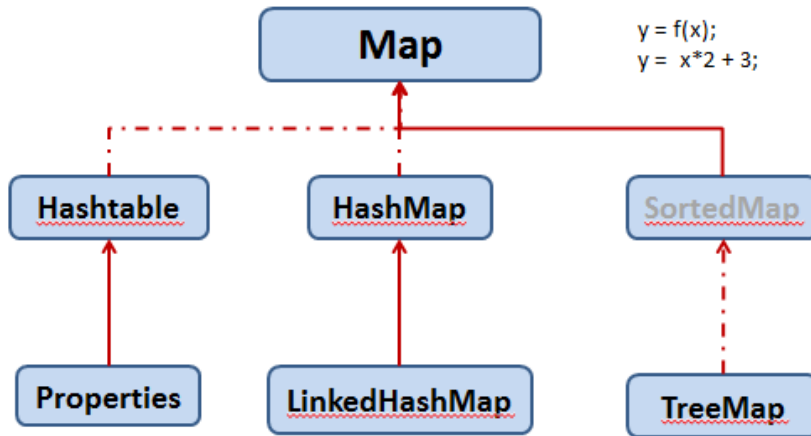
- Java集合框架
- Collection接口API
- Iterator迭代器接口
- Collection子接口之一：Set接口
 - [HashSet](#) [LinkedHashSet](#) [TreeSet](#)
- Collection子接口之二：List接口
 - [ArrayList](#) [LinkedList](#) [Vector](#)
- Map接口
 - [HashMap](#) [TreeMap](#) [Hashtable](#)
- Collections工具类

1. 主要有 Collection 和 Map 接口（体系里的实线表示 extends 关系，虚线表示 implements 关系）



以及

Map接口继承树



$y = f(x);$
 $y = x^2 + 3;$

7.1 Connection 接口的常用方法

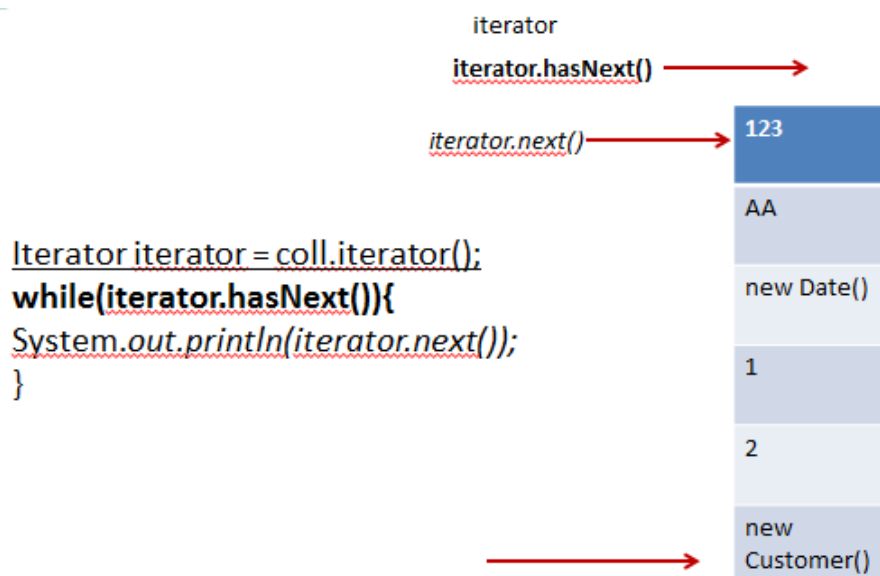
Collection 接口方法

boolean	add(E e) 确保此 collection 包含指定的元素（可选操作）。
boolean	addAll(Collection<? extends E> c) 将指定 collection 中的所有元素都添加到此 collection 中（可选操作）。
void	clear() 移除此 collection 中的所有元素（可选操作）。
boolean	contains(Object o) 如果此 collection 包含指定的元素，则返回 true。
boolean	containsAll(Collection<?> c) 如果此 collection 包含指定 collection 中的所有元素，则返回 true。
boolean	equals(Object o) 比较此 collection 与指定对象是否相等。
int	hashCode() 返回此 collection 的哈希码值。
boolean	isEmpty() 如果此 collection 不包含元素，则返回 true。

Iterator<E>	iterator() 返回在此 collection 的元素上进行迭代的迭代器。
boolean	remove(Object o) 从此 collection 中移除指定元素的单个实例，如果存在的话（可选操作）。
boolean	removeAll(Collection<?> c) 移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）。
boolean	retainAll(Collection<?> c) 仅保留此 collection 中那些也包含在指定 collection 的元素（可选操作）。
int	size() 返回此 collection 中的元素数。
Object[]	toArray() 返回包含此 collection 中所有元素的数组。
<T> T[]	toArray(T[] a) 返回包含此 collection 中所有元素的数组；返回数组的运行时类型与指定数组的运行时类型相同。

集合与数组间转换操作

7.2 迭代器 Iterator 的使用:



- 1.用于实现集合（Set、List、Map）元素的遍历
- 2.通过集合类的 `iterator()`方法返回一个 `Iterator` 实现类的对象，用来遍历调用 `iterator()`方法的集合的元素
- 3.步骤:

```
Collection coll = new HashSet();
coll.add().....//添加诸多个元素
Iterator iterator = coll.iterator();
while(iterator.hasNext()){
    System.out.println(iterator.next());
}
```

7.3 Connection 的子接口: Set

- 1.存储的元素是无序的、不可重复的！类似于高中概念中的集合。
- 2.主要实现类: `HashSet` `LinkedHashSet` `TreeSet`
- 3.`HashSet` 是 `Set` 的主要实现类 （可以添加 `null` 值，线程不安全的）
 - 3.1 在不使用泛型的时候，创建 `HashSet` 可以用来存储任何数据类型的数据，即 `add(Object obj)`;
 - 3.2 如何判断添加元素的不可重复性？（重点、难点）
标准：1.如果添加的 `JDK` 提供给我们的类的对象，会自动判断是否重复

2.如果是添加自定义类的对象，自定义类一定要重写

`equals()`和 `hashCode()`

注：当两个对象的 `equals()` 方法比较返回 `true` 时，这两个对象的 `hashCode()` 方法的返回值也应相等

4.添加元素的时候如何使用 `hashCode` 和 `equals()`方法的？

4.1 先调用 `hashCode` 方法获取对象的 `hash` 值。如果 `hash` 值跟之前的对象值都不同，直接存储

4.2 如果 `hash` 值与之前的某个对象相同，那么再通过 `equals()`方法进行比较。如果 `equals` 返回 `true`，添加不成功。返回 `false`，添加成功

5.`LinkedHashSet` ①遍历时，是按照添加元素的顺序实现的。②是可以添加任何数据类型的。③遍历时，效率比较高

6.`TreeSet` 存入的元素需要排序，遍历时，按照指定的顺序遍历。

关键点：`TreeSet` 中只能添加相同类型的数据。

6.1 自然排序

1.对于自定义类，此类要实现 `Comparable` 接口，重写此接口的 `CompareTo (Object obj)` 方法

2.创建 `TreeSet` 对象，使用 `add()`方法添加元素即可

注意点：当需要把一个对象放入 `TreeSet` 中，重写该对象对应的 `equals()` 方法时，应保证该方法与 `compareTo(Object obj)` 方法有一致的结果：如果两个对象通过 `equals()` 方法比较返回 `true`，则通过 `compareTo(Object obj)` 方法比较应返回 0

6.2 定制排序

1.创建一个实现 `Comparator` 接口的匿名对象

2.将此对象作为形参传递给 `TreeSet` 的构造器形参部分

3.创建 `TreeSet` 对象，使用 `add()`方法添加元素即可

注意点：此时添加对象如果是自定义的，无需再实现 `Comparable` 接口

7.4 `Connection` 的子接口： `List`

1.存储的数据是可重复的，是有序的。相当于可变长度的数组

2.实现类： `ArrayList` `LinkedList`（实现添加、删除、修改，效率高） `Vector`
（线程安全，但是是个古老的实现类，现在不常用了）

3.典型实现类： `ArrayList` （线程不安全的）

```
List list = new ArrayList();
```

4.常用的三个方法： ① `add(Object obj)` ② `Object get(int index)` ③ `int`

size()

5.使用 Iterator 迭代器实现遍历

7.5Map 接口及常用方法

1.存储的数据是以键值对 (key-value) 的形式存在的。类似于高中的函数

2.实现类: HashMap LinkedHashMap TreeMap Hashtable(古老类, 将其子类: Properties)

3.HashMap 是 Map 接口的主要实现类 Map map = new HashMap();

4.常用方法: map.put(key,value);添加一个元素
map.size();获取键值对的个数
map.get(key):获取指定 key 的 value 值

5.Map 实现类的遍历 (重点、难点)

Set keySet()

Collection values()

Set entrySet()

5.1 遍历 key 集

```
Set set = map.keySet();
```

```
for(Object obj : set){  
    System.out.println(obj);  
}
```

5.2 遍历 value 集

```
Collection coll = map.values();
```

```
Iterator i = coll.iterator();
```

```
while(i.hasNext()){  
    System.out.println(i.next());  
}
```

5.3 遍历 key-value 对集

(法一)

```
Set set1 = map.keySet();
```

```
for(Object obj : set1){  
    System.out.println(obj + "---->" + map.get(obj));  
}
```

(法二)

```
Set set2 = map.entrySet();
```

```
for(Object obj : set2){  
    Map.Entry entry = (Map.Entry)obj;  
    System.out.println(entry.getKey() + ":" + entry.getValue());  
}
```

6. LinkedHashMap: 按添加的顺序实现遍历

7. TreeMap : 遍历时, 按照添加元素时指定的顺序实现遍历。

要求, key 是一个 Set 集合。要求 key 必须实现自然排序或者是定制排序

8. Properties: 常用来操作属性文件。key 和 value 都是 String 型的

7.6 Collections 工具类

集合的工具类

可以用来操作: Set List Map

1. 对于集合元素的一般方法: reverse(list) sort(list) ..

2. 保证线程安全的同步方法: synchronizedXxx()

```
ArrayList list = new ArrayList();
```

```
list = Collections.synchronizedList(list);
```

-----第 8 章: 泛型-----

泛型 : JDK1.5 新加入的特性

使用: 1. 在集合中使用泛型

(在集合中使用泛型的声明和初始化)

```
Collection<String> coll = new ArrayList<String>();
```

```
Collection<Object> coll1 = new ArrayList<Object>();
```

```
Map<Integer,String> map = new HashMap<>();
```

```
Iterator<String> iterator = coll.iterator();
```

注意点: 1. 一旦声明泛型, 只能往里面添加声明类型的元素

2. 不同泛型的声明之间不能相互引用

3. 定义了泛型, 但是初始化时没有指明类型, 那么默认存入的是 Object 类型的元素

4. 声明泛型的接口或抽象类是不可以实例化的

5. 什么时候指明泛型类型? 当我们实例化泛型类时, 指明

泛型类型

(题外话: 数组的声明和初始化 String[] str = new String[4]; Integer i =

```
new Integer[]{21,3,5,67};)
```

2.自定义泛型类 泛型方法 泛型接口 (E: 是 Object 类或其子类, 不能是基本数据类型)

```
泛型类:    class 类名<E>{
            E e;
            public void set(E e){
                this.e = e;
            }
            public E getE(){
                return e;
            }
            public void show(){
                System.out.println(e);
            }
            //泛型方法:
            public <T> void fromArrayToCollection(T[] t,Collection<T> coll){
                for(T tt : t){
                    coll.add(tt);
                }
            }
        }
```

3.泛型跟继承的关系

List<Object> 、 List<Number> 、 List<Integer>三者之间没有关系!

4.通配符 <?>

List<Object> 、 List<Number> 、 List<Integer>都是 List<?>的子类

```
//public <? > void fromArrayToCollection(Collection<? > coll){
    //coll.add(tt);
//}
```


List<?> 只能够读取，不能够写入数据。（特殊：.add(null)）;

```
Set<Float> set = new Set<Float>;
```

```
    set.add(Float f);.....
```

```
Set<?> set1 = set;
```

```
for(Object obj : set1){
```

```
    System.out.println(obj);
```

```
}
```

5.有限制的通配符

<? extends Number> 只能够放入 Number 类或其子类类型的数据。

<? super Number> 只能够放入 Number 类或其父类类型的数据。

第 9 章：注解 & 枚举



一、枚举类

主要内容:

- 如何自定义枚举类
- 如何使用 `enum` 定义枚举类
 - 枚举类的主要方法
- 实现接口的枚举类

二、注解Annotation

主要内容

- JDK内置的基本注解类型（3个）
- 自定义注解类型
- 对注解进行注解（4个）
- 利用反射获取注解信息（在反射部分涉及）

1.枚举:

1.1 自定义枚举类 1.2 使用 enum 定义 1.3 values() valueOf(String str)

【典型代码 1】

```
class King{  
    private final String kingName;  
    private final String kingDesc;  
    //枚举类的对象  
    public static final King KING1 = new King("刘德华","拼命三郎");  
    public static final King KING2 = new King("郭富城","舞神");  
    public static final King KING3 = new King("黎明","长得帅");  
    public static final King KING4 = new King("张学友","歌神");  
  
    private King(String name,String desc){  
        this.kingName = name;  
        this.kingDesc = desc;  
    }  
}
```

```
public String getKingName() {  
    return kingName;  
}
```

```
public String getKingDesc() {  
    return kingDesc;  
}
```

```
@Override  
public String toString() {  
    return "King [kingName=" + kingName + ", kingDesc=" + kingDesc + "];"  
}
```

【典型代码 2】

```
enum King{  
    //枚举类的对象  
    KING1("刘德华","拼命三郎"),  
    KING2("郭富城","舞神"),  
    KING3("黎明","长得帅"),  
    KING4("张学友","歌神");  
    private final String kingName;  
    private final String kingDesc;  
  
    private King(String name,String desc){  
        this.kingName = name;  
        this.kingDesc = desc;  
    }  
  
    public String getKingName() {  
        return kingName;  
    }  
}
```

```

    }

    public String getKingDesc() {
        return kingDesc;
    }

    @Override
    public String toString() {
        return "King [kingName=" + kingName + ", kingDesc=" + kingDesc + "];"
    }

}

```

2.Annotation 元数据

2.1 JDK 内置的基本注解类型（3 个）

@Override: 限定重写父类方法, 该注释只能用于方法

@Deprecated: 用于表示某个程序元素(类, 方法等)已过时

@SuppressWarnings: 抑制编译器警告

2.2 自定义注解类型(仿照 suppresswarnings 写)

2.3 对注解进行注解（4 个）：元 Annotation

JDK5.0 提供了专门在注解上的注解类型，分别是：

Retention

Target

Documented

Inherited

- **java.io.File**类的使用
- IO原理及流的分类
- 文件流
 - FileInputStream / FileOutputStream / FileReader / FileWriter
- 缓冲流
 - BufferedInputStream / BufferedOutputStream /
 - BufferedReader / BufferedWriter
- 转换流
 - InputStreamReader / OutputStreamWriter
- 标准输入/输出流
- 打印流（了解）
 - PrintStream / PrintWriter
- 数据流（了解）
 - DataInputStream / DataOutputStream
- 对象流 ----涉及序列化、反序列化
 - ObjectInputStream / ObjectOutputStream
- 随机存取文件流
 - RandomAccessFile

1.File : 所代表的抽象路径，既可以是一个文件，可以一个是文件目录

1.1 File file = new File(String name);

File 类

访问文件名:

- getName()
- getPath()
- getAbsoluteFile()
- getAbsolutePath()
- getParent()
- renameTo(File
newName)

文件操作相关

- createNewFile()
- delete()

文件检测

- exists()
- canWrite()
- canRead()
- isFile()
- isDirectory()

➢ 目录操作相关

- mkdir()
- mkdirs()
- list()
- listFiles()

获取常规文件信息

- lastModified()
- length()

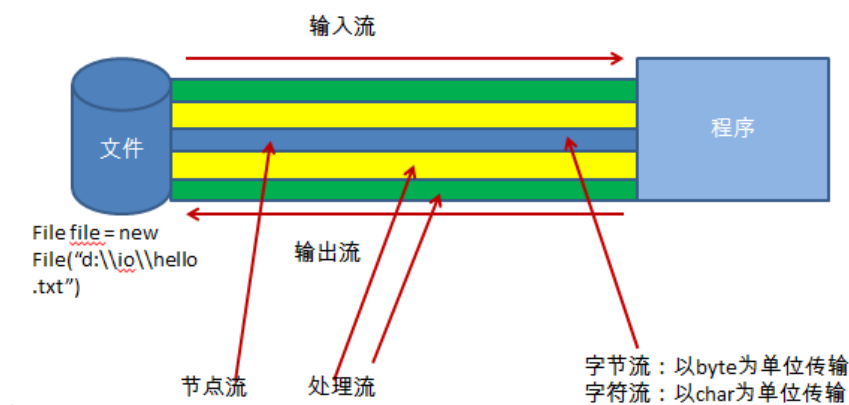
2. IO 流

2.1 IO 流的分类:

按操作数据单位不同分为: 字节流(8 bit), 字符流(16 bit)

按数据流的流向不同分为: 输入流, 输出流

按流的角色不同分为: 节点流, 处理流



以及, IO 流的体系

| 分类 | 字节输入流 | 字节输出流 | 字符输入流 | 字符输出流 |
|-------|----------------------|-----------------------|-------------------|--------------------|
| 抽象基类 | InputStream | OutputStream | Reader | Writer |
| 访问文件 | FileInputStream | FileOutputStream | FileReader | FileWriter |
| 访问数组 | ByteArrayInputStream | ByteArrayOutputStream | CharArrayReader | CharArrayWriter |
| 访问管道 | PipedInputStream | PipedOutputStream | PipedReader | PipedWriter |
| 访问字符串 | | | StringReader | StringWriter |
| 缓冲流 | BufferedInputStream | BufferedOutputStream | BufferedReader | BufferedWriter |
| 转换流 | | | InputStreamReader | OutputStreamWriter |
| 对象流 | ObjectInputStream | ObjectOutputStream | | |
| | FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| 打印流 | | PrintStream | | PrintWriter |
| 推回输入流 | PushbackInputStream | | PushbackReader | |
| 特殊流 | DataInputStream | DataOutputStream | | |

2.2 抽象基类

节点流（重点）

缓冲流（重点）

| | | | |
|--------------|------------------|--------|----------------------|
| InputStream | FileInputStream | -----> | BufferedInputStream |
| OutputStream | FileOutputStream | -----> | BufferedOutputStream |
| Reader | FileReader | -----> | BufferedReader |
| Writer | FileWriter | -----> | BufferedWriter |

2.3

【典型代码 1】

```
public void testFileInputStream(){
    InputStream is = null;
    try {
        File file = new File("Sat.txt");
        is = new FileInputStream(file);
        byte[] b = new byte[1024];
        int len;
        //
        while((len = is.read(b)) != -1){
            System.out.print(new String(b,0,len));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        try {
            is.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
}
```

【典型代码 2】

```
public void testReaderWriter() throws IOException{  
    Reader reader = new FileReader("dbcp.txt");  
    Writer writer = new FileWriter("dbcp1.txt");  
    char[] c = new char[1024];  
    int len = 0;  
    while((len = reader.read(c))!= -1){  
        writer.write(new String(c,0,len));  
        writer.flush();  
    }  
  
    writer.close();  
    reader.close();  
}
```

【典型代码 3】

//实现文件（文本文件、图片、视频等等）的复制

```
public static void copyFile(String src ,String des){  
    FileInputStream fis = null;  
    FileOutputStream fos = null;  
    try {  
        fis = new FileInputStream(new File(src));  
        fos = new FileOutputStream(new File(des));  
        byte[] b = new byte[1024];  
        int len;  
        while ((len = fis.read(b)) != -1) {  
            fos.write(b, 0, len);  
        }  
    } catch (IOException e) {
```



```

        e.printStackTrace();
    } finally {
        try {
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                fis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
}

```

【经典代码 4】

```

@Test
public void BufferedReaderWriter() throws IOException{
    Reader reader = new FileReader("dbcp.txt");
    Writer writer = new FileWriter("dbcp3.txt");
    BufferedReader br = new BufferedReader(reader);
    BufferedWriter bw = new BufferedWriter(writer);
    //BufferedReader br = new BufferedReader(new FileReader("dbcp.txt"));
    //BufferedWriter bw = new BufferedWriter(new FileWriter("dbcp1.txt"));
    //    char[] c = new char[1024];
    String str = null;
    //int len = 0;
    while((str = br.readLine())!= null){
        bw.write(str);
    }
}

```

```

        bw.newLine();
        bw.flush();
    }

    bw.close();
    br.close();
}

```

【经典代码 5】使用缓冲流实现文件（文本文件、图片、视频等等）的复制。并通过实例比较此代码与【经典代码 3】的效率

```

public static void copyBufferedInputOutput(String src, String des)
    throws IOException {
    File file1 = new File(src);
    File file2 = new File(des);
    // 2.
    FileInputStream fis = new FileInputStream(file1);
    FileOutputStream fos = new FileOutputStream(file2);
    // 3.将 FileInputStream 的节点流对象作为形参传递给 BufferedInputStream 构造器。
    BufferedInputStream bis = new BufferedInputStream(fis);
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    // 4.实现复制操作的细节
    byte[] b = new byte[1024];
    int len;
    while ((len = bis.read(b)) != -1) {
        bos.write(b, 0, len);
        bos.flush();
    }

    bos.close();
    bis.close();
}

```

```
}
```

2.4 转换流： 实现从字节流到字符流的转换： `InputStreamReader` 和 `OutputStreamWriter`

```
@Test
```

```
public void testConvertStream() throws IOException{
    Reader r = new InputStreamReader(new FileInputStream("dbcp.txt"), "GBK");
    Writer w = new OutputStreamWriter(new
FileOutputStream("dbcp4.txt"),"GBK");
    BufferedReader br = new BufferedReader(r);
    BufferedWriter bw = new BufferedWriter(w);
    String str = null;
    while((str = br.readLine()) != null){
        bw.write(str);
        bw.newLine();
        bw.flush();
    }

    bw.close();
    br.close();
    w.close();
    r.close();
}
```

2.5 标准输入输出流、数据流、打印流（了解）

2.6 对象流： `ObjectInputStream` 和 `ObjectOutputStream` 可以用来读取和写入基本数据类型的数据以及 JDK 提供类、自定义类的对象。

2.6.1 `ObjectInputStream` 对应有 `readObject()`方法：对象的反序列化

`ObjectOutputStream` 对应有 `writeObject(Object obj)`方法：对象的序列化

2.6.2 要求存储的对象对应的类一定要实现序列化的机制（implements

Serializable) 同时，要求类的属性对应的类也必须实现序列化。

>凡是实现 Serializable 接口的类都有一个表示序列化版本标识符的静态变量：`private static final long serialVersionUID = 12L;`

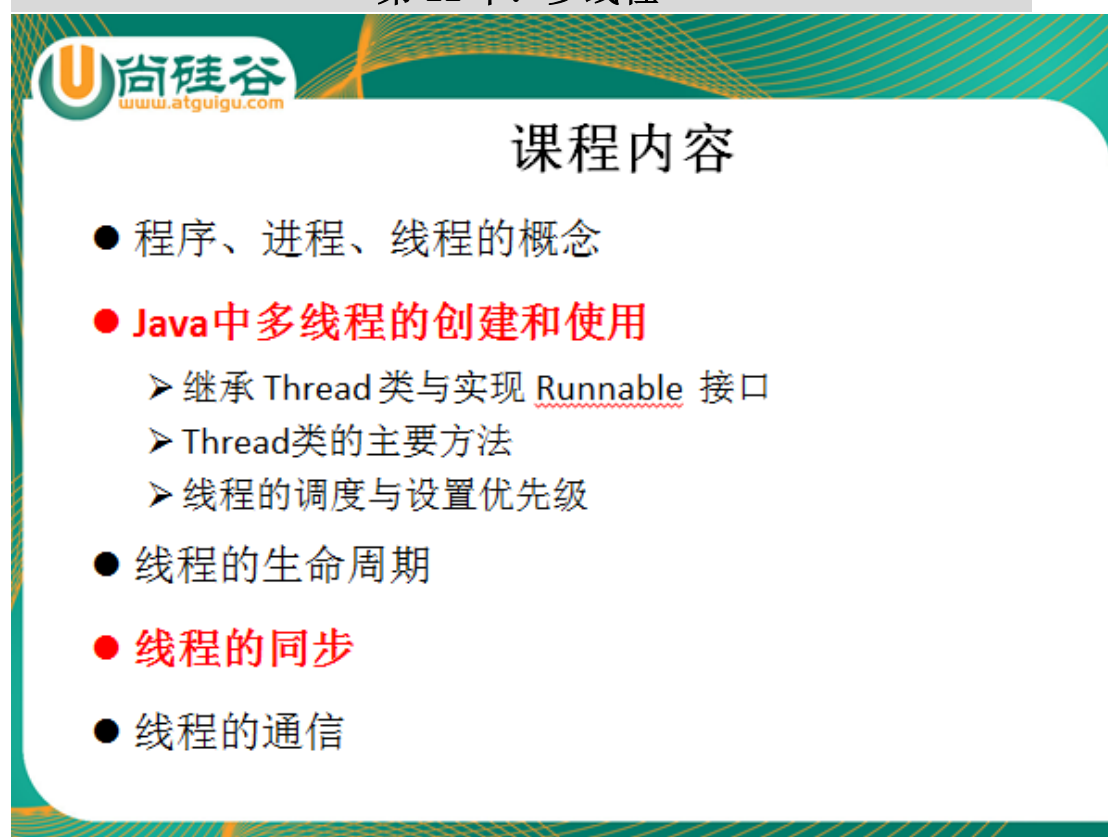
>ObjectOutputStream 和 ObjectOutputStream 不能序列化 static 和 transient 修饰的成员变量

2.7 RandomAccessFile :支持随机读、取文件，一定有 read 方法和 write 方法

>对于文件的写入，支持文件的开头、中间位置的插入，以及末位的写入。

注意：插入操作实现的对源文件对应位置的“覆盖”，而非“插入”。

第 11 章：多线程



尚硅谷
www.atguigu.com

课程内容

- 程序、进程、线程的概念
- **Java中多线程的创建和使用**
 - 继承 Thread 类与实现 Runnable 接口
 - Thread类的主要方法
 - 线程的调度与设置优先级
- 线程的生命周期
- **线程的同步**
- 线程的通信

1. 程序 进程 线程 (了解)

2. 如何实现多线程的创建 (两种方式) 和启动

1.继承 Thread 类

1) 子类继承 Thread 类

2) 重写 Thread 类的 run()方法? run()中是创建的多线程要实现的功能。

3) 创建子类的对象 (创建了几个, 就有几个线程)

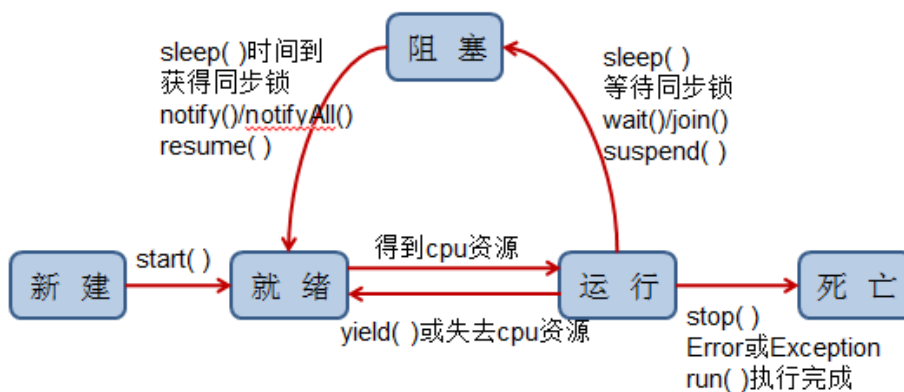
4) 启动: 对象.start();

2.实现 Runnable 接口

- 1) 子类实现 Runnable 接口
 - 2) 重写 Runnable 接口的 run()方法?
 - 3) 创建一个子类的实例
 - 4) 将子类的实例作为实参传递给 Thread 的构造器中?
 - 5) 启动: 对象.start();
- >比较两种方式的区别: 联系, 不同点, 哪个好

3.涉及 Thread 类的几个方法:

4.多线程的生命周期:



线程状态转换图

5.线程的同步 (解决多线程的安全问题)

前提: 多个线程操作共享数据

解决方法: 在其中一个线程 A 对共享数据修改的时候, 其它线程必须在外边等候, 当 A 线程执行完共享数据后, 其它线程方可参与执行共享数据。

5.1 同步代码块

1.synchronized(对象){

```

*    //涉及共享数据的代码
* }
*

```

5.2 同步方法

```

* 2.public synchronized void show(){
*    //涉及共享数据的代码
* }

```

>对于非 static 的同步方法, 它对应的锁是 this, 即当前对象。

>对于 static 的同步方法, 它对应的锁是此方法所在的类。 XxxYyy.class

5.3 死锁 (写程序时, 需要避免的)

不同的线程分别占用对方需要的同步资源不放弃, 都在等待对方放弃自己需要的

同步资源，就形成了线程的死锁

6.线程通信

wait() notify() notifyAll() 一定使用在同步代码块或同步方法中。

6.1 会释放同步锁的操作

6.2 不会释放同步锁的操作

-----第 12 章：Java 常用类 -----



1.String 类 StringBuffer 类 StringBuilder 类

String 类：不可变的字符串序列

1.常用的方法

2.String 类与基本数据类型（或包装类）的转换

1) 字符串---->包装类： 如：Integer.parseInt(String str) 2.包装类 ---->字符串

串：String.valueOf(Xxx xxx)

3.String 类与字符数组、字节数组的转换

* 1.字节数组---->字符串： String str = new String(byte[] b);

* 2.字符串 ---->字节数组: `byte[] b = str.getBytes();`

* 3.字符数组 ---->字符串: `String str = new String(char[] c);`

* 4.字符串 ----> 字符数组: `char[] c = str.toCharArray();`

`StringBuffer` 是可变的字符串序列

1.相比 `String` 类, 多了一些方法: `append()` `reverse()` `insert` `replace()`....

`StringBuilder` 是可变的字符串序列 :JDK1.5 新加的, 处理字符串的效率更高, 线程不安全的。

>三者的效率测试 `v(StringBuilder) > v(StringBuffer) > v(String)`

2.日期类相关的

2.1 `System.currentTimeMillis();`

2.2 `Date` 类 `java.util` 包下 `Date date = new Date();` `date.getTime();`

2.3 `SimpleDateFormat` 类 1.格式化: 日期 `Date` ---->文本 2.解

析: 文本---->日期 `Date`

2.4 `Calendar` 类

3.Math `BigInteger` `BigDecimal`

-----第 13 章: Java 反射 -----

课程内容

- 1.理解Class类并实例化Class类对象
- 2.运行时创建类对象并获取类的完整结构
- 3.通过反射调用类的指定方法、指定属性
- 4.动态代理

13.1 理解类的加载过程

任何一个类编译后生成一个.class文件，JVM的类加载器将此.class文件加载到内存中。.class文件就对应有一个java.lang.Class类的实例。

可以理解为，加载的类本身就作为Class类的实例

> 涉及到的包/类：① java.lang.Class 可以看做反射的根源 ②

java.lang.reflect

13.2 如何实例化 Class 类（掌握）

//1.直接调用类的属性.class

```
Class clazz = Person.class;
```

```
clazz = Animal.class;
```

//2.调用Class类的静态方法forName(全类名),可能抛ClassNotFoundException异常

```
Class clazz2 = Class.forName("com.atguigu.exer.Animal");
```

```
System.out.println(clazz2);
```

//3.通过运行时类的对象的getClass()方法获取


```
Person p = new Person();
Class clazz3 = p.getClass();
System.out.println(clazz3);
```

//4.了解

```
ClassLoader classLoader = this.getClass().getClassLoader();
Class clazz4 = classLoader.loadClass("com.atguigu.exer.Animal");
```

13.3 通过 Class 对象可以创建运行时类（即 Class 对象对应的类）的对象，以及获取运行时类完整的类结构

3.1) Person p = (Person)clazz.newInstance();

```
Constructor constructor =
clazz.getDeclaredConstructor(String.class,Integer.class);
Person p1 = (Person)constructor.newInstance("小明",23);
```

3.2)通过 Class 对象获取类的完整结构：包名、实现的接口、父类、带泛型的父类、注解（类上、属性、方法、构造器）、属性、方法、构造器、内部类以及对应的属性的修饰符、声明的类型、名字；方法的修饰符、返回值类型、方法名、形参列表

13.4 创建运行时类的对象，并获取、调用指定的类结构

【典型代码 1】创建运行时类的对象，并获取指定的属性为其赋值

```
@Test
public void getOneField() throws Exception{
    //1.创建运行时类的对象
    Class clazz = Person.class;
    Person p = (Person)clazz.newInstance();
    System.out.println(p);
```

```

//2.获取运行时类对象的属性
Field field = clazz.getField("name");
//如果属性声明为 private,那么只能 getDeclaredField(String str)的方式调用
Field field1 = clazz.getDeclaredField("age");
//3.给运行时类的对象的相应属性赋值
//field.set(Object obj,赋值 a): 给 obj 的对象对应的 field 属性赋值为 a。
field.set(p, "小丽");
//当试图对声明为 private 的属性进行修改时,必须先调用属性的
setAccessible(true), 方可对属性进行修改(赋值)
field1.setAccessible(true);
field1.set(p, 22);
//4.输出此对象
System.out.println(p);
}

```

【典型代码 2】创建运行时类的对象,并获取指定的方法(考虑给方法的形参赋值)供对象调用。

@Test

```

public void getOneMethod() throws Exception{
//1.创建运行时类的对象
Class clazz = Person.class;
Person p = (Person)clazz.newInstance();
//2.获取运行时类对象的方法
Method method = clazz.getMethod("display", String.class,Integer.class);
Method method1 = clazz.getDeclaredMethod("show", null);
//3.调用运行时类的对象的方法,同时若方法有形参,需要给形参赋值。
//invoke(Object obj,Object...args):给 obj 对象相应方法的形参赋值为
Object...args
method.invoke(p, "张三",33);
//System.out.println(method);
}

```


InetAddress.getByName("192.168.4.32");

14.2.2 调用 InetAddress 的 getLocalHost()方法，获取本机的 ip 地址。

14.2.3 主要的方法：getHostName(): 获取域名 getHostAddress(): 获取 IP 地址

14.3 网络协议

网络通信协议

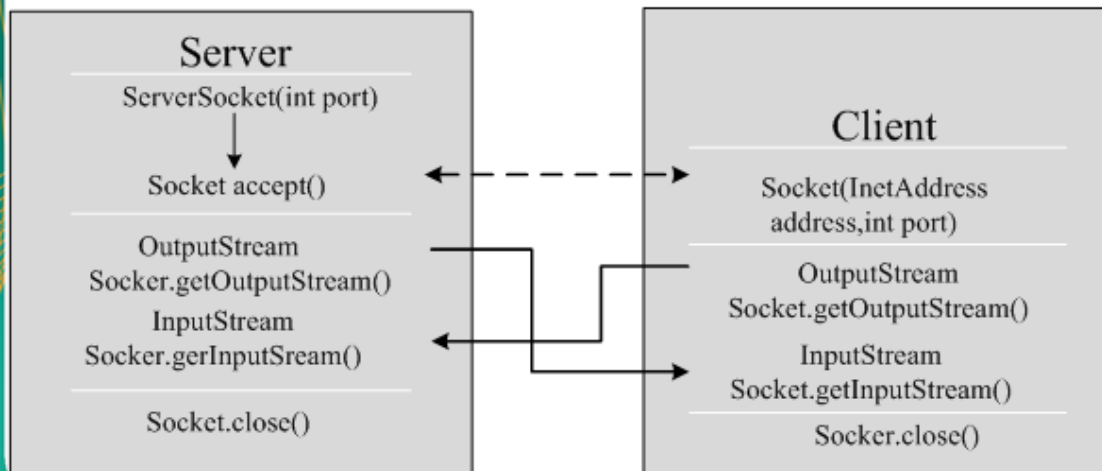
| OSI参考模型 | TCP/IP参考模型 | TCP/IP参考模型各层对应协议 |
|---------|------------|------------------------|
| 应用层 | 应用层 | HTTP、ftp、telnet、DNS... |
| 表示层 | | |
| 会话层 | | |
| 传输层 | 传输层 | TCP、UDP、... |
| 网络层 | 网络层 | IP、ICMP、ARP... |
| 数据链路层 | 物理+数据链路层 | Link |
| 物理层 | | |

14.4 TCP 协议编程

安全，效率比较低；客户端&服务端；通过 IO 流的形式，实现客户端和服务端的交互

基于Socket的TCP编程

- Java语言的基于套接字编程分为服务端编程和客户端编程，其通信模型如图所示：



基于TCP的Socket通信

实现客户端和服务端的编程：

1) 客户端: ① Socket socket = new Socket(InetAddress address, int port);
或者

Socket socket = new Socket(String host, int port);

② 调用 socket 对象的 getInputStream() 和 getOutputStream() 方法实现与服务端的传输

③ 关闭相应的流和 socket 对象

2) 服务端: ① ServerSocket serverSocket = new ServerSocket(int port);

② Socket socket1 = severSocket.accept();

③调用 socket1 对象的 getInputStream() 和 getOutputStream() 方法实现与客户端的传输、

④关闭相应的流和 socket 对象、serverSocket 对象

14.5UDP 协议编程

不安全, 效率高; 传输端&接收端; 通过 IO 流的形式, 实现客户端和服务端的交互

①创建传输端和接收端的对象 ②对于传输端, 如何创建数据报: DatagramPacket ,包含着要传输的内容, 以及接收端的 ip 和端口号。

【典型代码】

@Test

```
public void testSend() throws IOException{
    DatagramSocket ds = new DatagramSocket();
    byte[] by = "hello,atguigu.com".getBytes();
    DatagramPacket dp = new DatagramPacket(by,0,by.length,
        InetAddress.getByName("127.0.0.1"),10000);
    ds.send(dp);
    ds.close();
}
```

@Test

```
public void testReceive() throws IOException{
    DatagramSocket ds = new DatagramSocket(10000);
    byte[] by = new byte[1024];
    DatagramPacket dp = new DatagramPacket(by,by.length);
    ds.receive(dp);
    String str = new String(dp.getData(),0,dp.getLength());
    System.out.println(str+"--"+dp.getAddress());
    ds.close();
}
```

14.6URL(Uniform Resource Locator)

统一资源定位符, 它表示 Internet 上某一资源的地址。

1.URL url = new URL("http://localhost:8080/examples/myTest.txt");

2. URL 类常用的方法

| | |
|------------------------------|-------------------|
| public String getProtocol() | 获取该 URL 的协议名 |
| public String getHost() | 获取该 URL 的主机名 |
| public String getPort() | 获取该 URL 的端口号 |
| public String getPath() | 获取该 URL 的文件路径 |
| public String getFile() | 获取该 URL 的文件名 |
| public String getRef() | 获取该 URL 在文件中的相对位置 |
| public String getQuery() | 获取该 URL 的查询名 |

3. 如果仅需要读取网络资源: URL 的方法 `openStream()`: 能从网络上读取数据

如果希望将网络资源存储在本地: ① 调用 URL 的 `openConnection()` 方法, 返回一个 `URLConnection` 对象

②调用 `URLConnection` 对象的 `getInputStream()`

方法

【典型代码】

```
//获取 URL 对象的资源
```

```
//1.openConnection 的方法, 获取 URLConnection 的对象
```

```
URLConnection urlconn = url.openConnection();
```

```
//2.调用 URLConnection 的 getInputStream()方法, 获取输入流
```

```
InputStream is = urlconn.getInputStream();
```

```
OutputStream os = new FileOutputStream(new File("myTest1.txt"));
```

```
//3.实现细节
```

```
byte[] b= new byte[1024];
```

```
int len;
```

```
while((len = is.read(b)) != -1){
```

```
    os.write(b, 0, len);
```

```
}
```