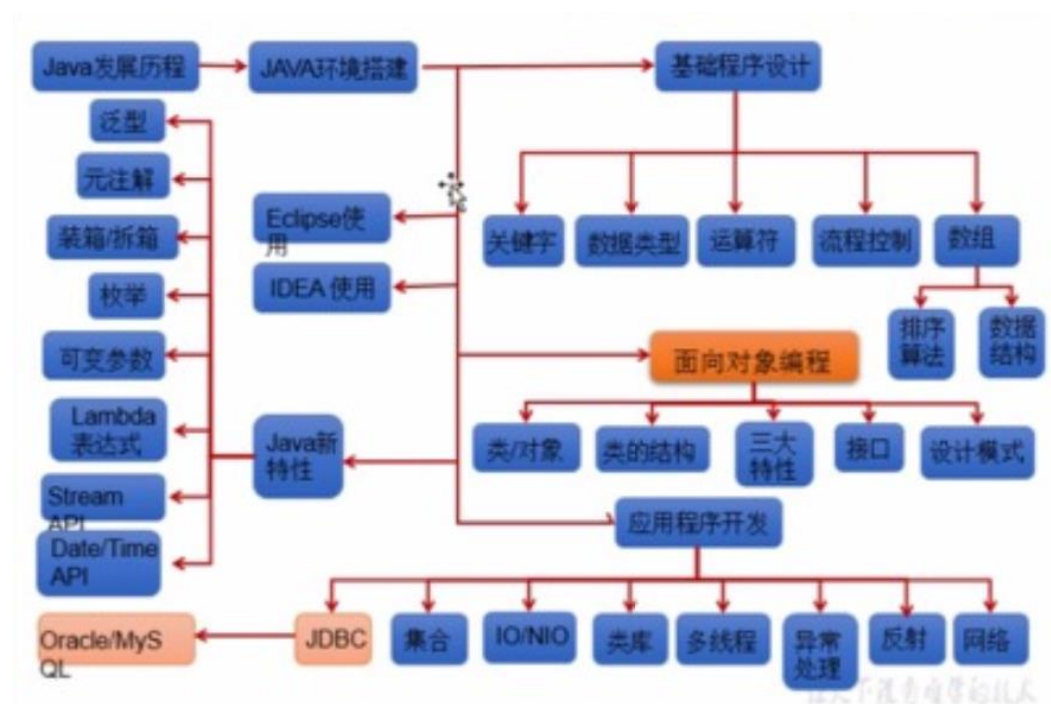


1 第一天：P1-P20 计算机介绍、Java 简介、JDK JRE

1.1 Java 学习路线



1.2 常用 Dos 命令

● 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

✓ 补充：echo javase>1.txt

将内容javase写入到1.txt文件中，如果该文件不存在则直接创建

● 常用快捷键

- **← →**：移动光标
- **↑ ↓**：调阅历史操作命令
- **Tab**：补充文件或目录的名字
- **Delete**和**Backspace**：删除字符

1.3 Java 语言版本迭代

1995 年发布 JDK1.0；JDK1.5 开始改名为 JDK5.0,以后都是 6.0, 7.0 ...。

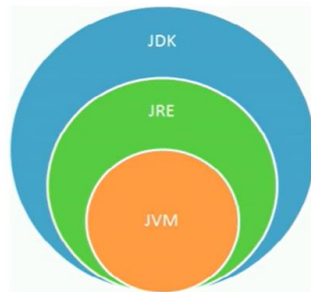
JavaSE (J2SE)：Java 核心 API + 桌面级应用开发，不如 Qt 和 C#

JavaEE (J2EE)：企业级开发 (Web 应用)

JavaME (J2ME)：移动端应用开发，被安卓取代了

1.4 Java 开发环境

1.4.1 JDK JRE JVM



- JDK = JRE + 开发工具集（例如Javac编译工具等）
- JRE = JVM + Java SE标准类库

1.4.2 环境变量配置

```
JAVA_HOME = D:\PC_Software_Install\Java\jdk1.8.0_131;
PATH = %JAVA_HOME%\bin;
```

1.5 Hello World 程序

- (1) 创建：创建一个以.java 结尾的文件
- (2) 编辑：

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

- (3) 编译：javac 源文件名.Java 可以生成一个或多个字节码文件（有几个类就有几个字节码文件）
- (4) 运行：java 字节码文件名（注意：不要.class 拓展名）

```
hello, world
请按任意键继续. . .
```

说明：

1. 一个源文件中可以由多个类，但是只能有一个 public 类；
2. public 类的名字必须和源文件名一样

1.6 注释

- (1) 单行注释：//
- (2) 多行注释：/* */
多行注释不能被嵌套使用
- (3) 文档注释：/** */

```
javadoc -d 目标路径 -author -version 源文件名.java
```

2 第二天：P21-P28 变量

2.1 变量的声明和赋值

第一种：声明变量，同时赋值

```
// 一个变量
int a = 1;
// 多个变量
int a = 1, b = 2, c = 3;
```

第二种：先声明后赋值

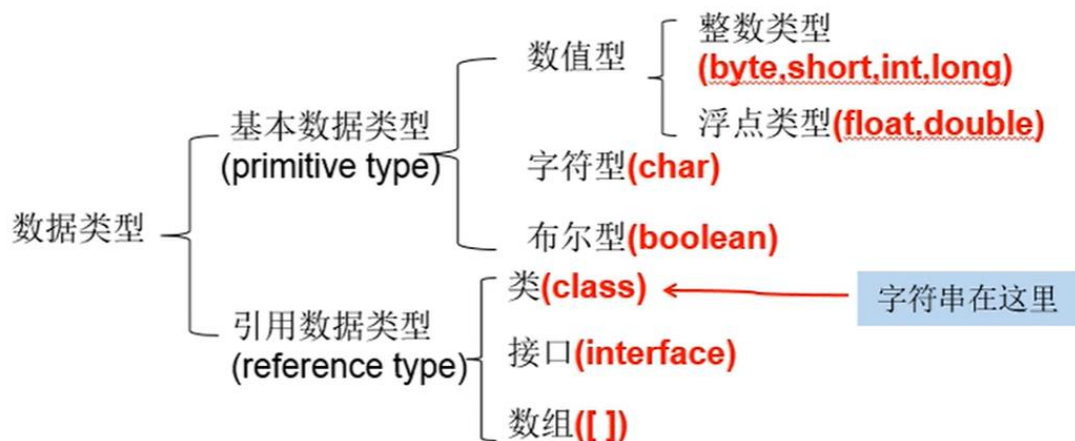
```
// 一个变量
int a;
a = 2;
// 多个变量
int a, b, c;
a = b = c = 10;
```

说明：

- 1、变量的作用域：声明变量所在的那对大括号内；
- 2、在同一个作用域内，变量名不允许重复；
- 3、变量应该先声明后使用；
- 4、同一个变量可以被多次赋值，后一次赋值覆盖前一次赋值。

2.2 Java 变量的分类

2.2.1 按数据类型分类：



(1) 整数类型：byte、short、int、long

Java 的整数类型有固定的表数范围和字段长度，不受具体的操作系统的影响，以保证 Java 程序的可移植性。

Java 整型常量默认为 int 类型，long 类型常量必须在末尾加上 “l” 或者 “L”。

类型	占用的存储空间	表数范围
byte	1 字节	-128 ~ 127

short	2 字节	-32768 ~ 32767
int	4 字节	-2147483648 ~ 2147483647
long	8 字节	$-2^{63} \sim 2^{63} - 1$

(2) 浮点类型：float、double

Java 的浮点类型有固定的表数范围和字段长度，不受具体的操作系统的影响，以保证 Java 程序的可移植性。

Java 浮点型常量默认为 double 类型（也可以在末尾加上“d”或者“D”），float 类型常量必须在末尾加上“f”或者“F”。

浮点数表示形式：

十进制数表示法：5.12, 512.0f, .512

科学计数表示法：5.12e2, 512E2, 100E-2

类型	占用的存储空间	表数范围
float	4 字节	-3.403E38 ~ 3.403E38
double	8 字节	-1.798E308 ~ 1.798E308

(3) 字符类型：char

Java 中的字符类型是占两个字节的。其表示方式有以下三种：

```
// 1、单个普通字符
char c1 = 'A';
char c2 = '中';
// 2、转义字符
char n1 = '\n';
// 3、Unicode 值
char n2 = '\u0056';
char c3 = 65;
```

表 2-1 转义字符

转义字符	意义	ASCII 码值（十进制）
\a	响铃(BEL)	007
\b	退格(BS)，将当前位置移到前一位	008
\f	换页(FF)，将当前位置移到下页开头	012
\n	换行(LF)，将当前位置移到下一行开头	010
\r	回车(CR)，将当前位置移到本行开头	013
\t	水平制表(HT)（跳到下一个 TAB 位置）	009
\v	垂直制表(VT)	011
\\	代表一个反斜线字符\"	092
\'	代表一个单引号（撇号）字符	039

\"	代表一个双引号字符	034
\?	代表一个问号	063
\0	空字符(NUL)	000
\ddd	1 到 3 位八进制数所代表的任意字符	三位八进制
\xhh	十六进制所代表的任意字符	十六进制

拓展:Unicode 编码与 UTF-8

UTF-8 是 Unicode 的实现方式之一。

UTF-8 的编码规则很简单，只有二条：

(1) 对于单字节的符号，字节的第一位设为 0，后面 7 位为这个符号的 Unicode 码。因此对于英语字母，UTF-8 编码和 ASCII 码是相同的。

(2) 对于 n 字节的符号 (n > 1)，第一个字节的前 n 位都设为 1，第 n + 1 位设为 0，后面字节的前两位一律设为 10。剩下的没有提及的二进制位，全部为这个符号的 Unicode 码。

Unicode 符号范围 (十六进制)	UTF-8 编码方式 (二进制)
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

根据上表，解读 UTF-8 编码非常简单。如果一个字节的第一位是 0，则这个字节单独就是一个字符；如果第一位是 1，则连续有多少个 1，就表示当前字符占用多少个字节。

引用自: http://www.ruanyifeng.com/blog/2007/10/ascii_unicode_and_utf-8.html

(4) 布尔类型

只有 true 和 false 两个值。

(5) 各类型所占字节数

```
/**
输出 Java 中基本类型所占字节数
版权声明：本文为 CSDN 博主「阳光岛主」的原创文章，遵循 CC 4.0 BY-SA 版权协议，
转载请附上原文出处链接及本声明。
原文链接：https://blog.csdn.net/ithomer/article/details/7310008
*/
public class CalSize {
    public static void main(String[] args) {
        System.out.println("int:\t" + Integer.SIZE/8);        // 4
        System.out.println("short:\t" + Short.SIZE/8);        // 2
        System.out.println("long:\t" + Long.SIZE/8);           // 8
        System.out.println("byte:\t" + Byte.SIZE/8);           // 1
        System.out.println("char:\t" + Character.SIZE/8);      // 2
    }
}
```

```

        System.out.println("float:\t" + Float.SIZE/8);        // 4
        System.out.println("double:\t" + Double.SIZE/8);    // 8
        //System.out.println("Boolean: " + Boolean);
    }
}

```

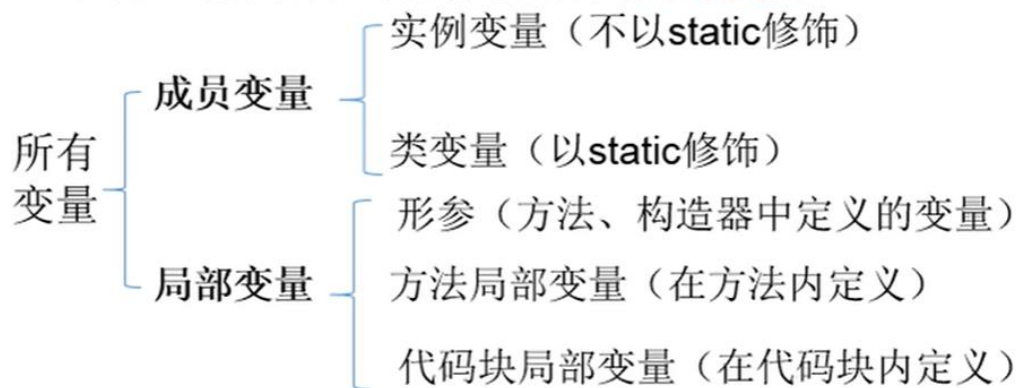
```

int:    4
short:  2
long:   8
byte:   1
char:   2
float:  4
double: 8

```

2.2.2 按声明的位置分类

- 在方法体外，类体内声明的变量称为**成员变量**。
- 在方法体内部声明的变量称为**局部变量**。



- **注意：二者在初始化值方面的异同：**

同：都有生命周期 异：局部变量除形参外，需显式初始化。

2.3 变量间的运算

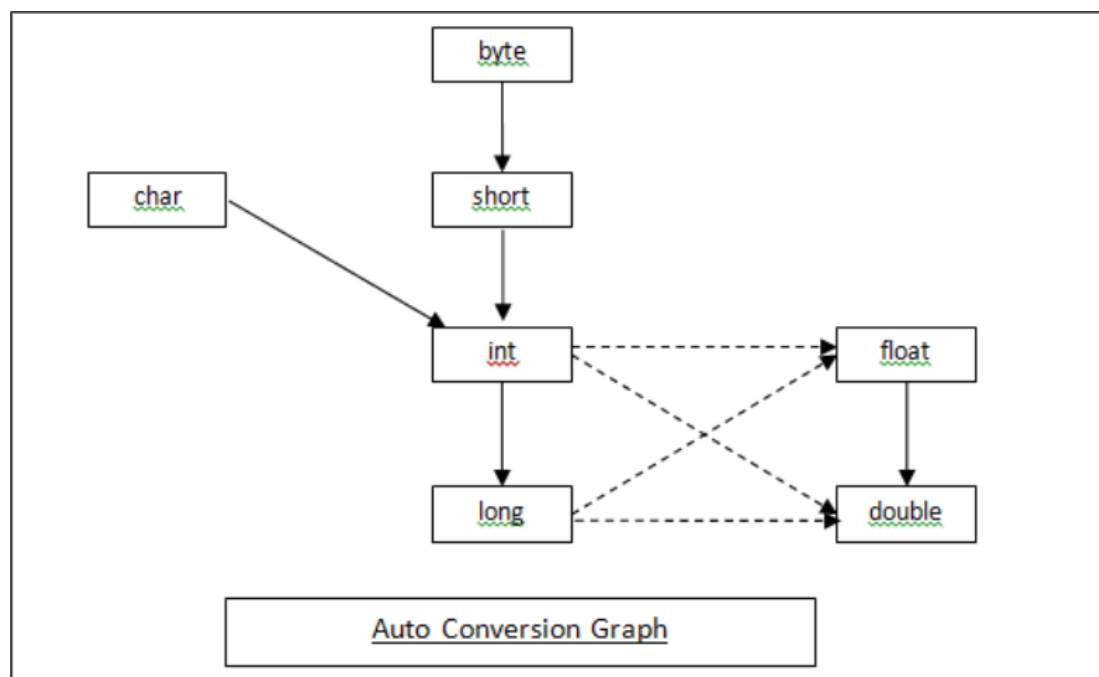
2.3.1 基本数据类型间的运算（不包括 boolean）

(1) 自动类型提升：小容量的变量与大容量的变量做运算，结果用大容量的变量来接收。

byte, short, char → int → long → float → double

注意：

- 1、容量指的是表数范围，而不是所占存储空间。float (4 字节) 的表数范围大于 long (8 字节)。
- 2、byte, short, char 三者之间的运算都会自动提升为 int 类型。



实线表示自动转换时不会造成数据丢失，虚线则可能会出现数据丢失问题。

这是字段类型提升的一个坑：

自动类型提升有好处，但它也会引起令人疑惑的编译错误。例如，下面看起来正确的程序却会引起问题：

```
byte b = 50;
b = b * 2;    // Type mismatch: cannot convert from int to byte
```

如上所示，第二行会报“类型不匹配：无法从 int 转换为 byte”错误。

Type. java:4: 错误：不兼容的类型：从int转换到byte可能会有损失
 b = b * 2; // Type mismatch: cannot convert from int to byte

1 个错误

该程序试图将一个完全合法的 byte 型的值 50*2 再存储给一个 byte 型的变量。但是当表达式求值的时候，操作数被自动的提升为 int 型，计算结果也被提升为 int 型。这样表达式的结果现在是 int 型，不强制转换它就不能被赋为 byte 型。确实如此，在这个特别的情况下，被赋的值将仍然适合目标类型。

所以应该使用一个显示的强制类型转换，例如：

```
byte b = 50;
b = (byte)(b * 2);
```

这样就能产生正确的值 100。

注意：char 类型比较特殊，char 自动转换成 int、long、float 和 double，但 byte 和 short 不能自动转换为 char，而且 char 也不能自动转换为 byte 或 short。

引用自：<http://c.biancheng.net/view/796.html>

这是另一个坑

float 和 long 运算，会自动类型提升为 float。例如如下代码编译错误：

```
long l = 200L;
float f = 10.0F;
```



```
long r = l + f;
```

Type. java:12: 错误: 不兼容的类型: 从float转换到long可能会有损失
long r = l + f;

1 个错误

需要使用 float 类型变量来存储运算结果。

```
long l = 200L;  
float f = 10.0F;  
float r = l + f;
```

(2) 强制类型转换: 自动类型提升的逆过程。

格式: 目标类型 变量 = (目标类型)源类型变量/常量

注意:

- 1、使用强制类型转换运算符“(类型)”。
- 2、可能会损失精度或者类型溢出。
- 3、在强制类型转换中目标类型和源类型变量的类型始终没有发生改变。

(3) 有关于 Java 常量

```
3  /*  
4  
5     java的整型常量默认为 int 型  
6     Java 的浮点型常量默认为double型  
7  */  
8  public class VarTest6{  
9  
10     public static void main(String[] args){  
11  
12         //Java 的浮点型常量默认为double型,所以float类型的值必须加f  
13         float f = 12.3f;  
14  
15         //java的整型常量默认为 int 型  
16         long l = 1234564654564L; //常量可以不加L那么就只能在int类型的表数范围内  
17  
18  
19         byte b = 20; //因为底层做了处理  
20  
21         System.out.println(b);  
22  
23     }  
24 }  
25 }
```

2.3.2 基本数据类型和 String 间的运算

- 1、字符串和基本数据类型之间只能做连接运算, 没有自动类型提升。
- 2、字符串做链接运算的结果是字符串类型, 只能用字符串来接受
- 3、注意“+”号是做加法, 还是做字符串连接符。

```
int num = 1;  
char cc = 'a';  
String st = "小苍苍";  
System.out.println(num + cc + st); //98小苍苍  
System.out.println(num + st + cc); //1小苍苍a  
System.out.println(st + cc + num); //小苍苍a1  
System.out.println(st + (cc + num)); //小苍苍98
```


3 第二天：P29-P30 关键字、保留字、标识符

3.1 关键字和保留字

关键字的定义：Java 关键字是 Java 语言里事先定义的，被赋予了特殊含义的标识符。

关键字的特点：关键字均为小写。

保留字的定义：现版本暂未使用，但以后版本可能用来作为关键字。

表 3-1 Java 关键字（51-3+2 个）

用于定义数据类型的关键字（12 个）：							
class	interface	enum	void				
byte	short	int	long	float	double	char	boolean
用于表示值的关键字（3 个）不认为是关键字							
true	false	null					
用于定义流程控制的关键字（11 个）							
if	else	switch	case	default			
while	do	for	break	continue	return		
用于定义访问权限的关键字（3 个）							
private	protected	public					
类与类或者接口之间关系的关键字（2 个）							
extends	implements						
用于类、函数、变量的修饰符的关键字（5 个）							
abstract	final	static	synchronized	volatile			
与对象实例有关的关键字（4 个）							
new	this	super	instanceof				
用于异常处理的关键字（5 个）							
try	catch	finally	throw	throws			
与包有关的关键字（2 个）							
package	import						
其他（4 个）							
native	strictfp	transient	assert				

表 3-2 Java 保留字（11 个）

byValue	cast	future	generic	inner	operator	outer	rest
var	goto	const					

goto 和 const 被认为是关键字

注：java 官方文档有说明：“An identifier cannot have the same spelling (Unicode character sequence) as a keyword (§ 3.9), boolean literal (§ 3.10.3), or the null literal (§ 3.10.7), or a compile-time error occurs. While true and false might appear to be keywords, they are technically boolean literals (§ 3.10.3). Similarly, while null might appear to be a keyword, it is technically the null literal (§ 3.10.7).”

【译文：标识符不能具有与关键字 (§ 3.9)，布尔文字 (§ 3.10.3) 或空文字 (§ 3.10.7) 相同的拼写 (Unicode 字符序列)，否则会发生编译时错误。尽管 true 和 false 可能是关键字，但从技术上讲，它们是布尔文字 (第 3.10.3 节)。同样，尽管 null 似乎是一个关键字，

但从技术上讲，它是 null 文字（第 3.10.7 节）。】

true false null 不属于关键字，但是属于标识符。规定的关键字只有 50 个，包含两个保留字(goto, const)，但是这 53 个都属于标识符。它们之间的关系是：标识符包含 关键字、boolean literal (true,false)、 null literal; 关键字里面又包含有两个保留字。

引用自：<https://blog.csdn.net/u012506661/article/details/52756452> 下的评论

以下是《Java 语言规范 基于 Java SE 8》一书中关于“关键字”的描述。（第 15 页）

3.9 关键字

Java 中保留了 50 个关键字，它们都是由 ASCII 字母构成的字符序列，并且不能当作标识符（第 3.8 节）使用。

Keyword: one of

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

尽管现在已经不再使用关键字 const 和 goto 了，但是仍旧保留了。这使得当这些 C++ 关键字在程序中使用不当时，Java 编译器能够产生更有用的错误消息。

尽管 true 和 false 看起来应该被当作关键字，但是从技术上讲，它们仅仅只是布尔字面常量（第 3.10.3 节）。与此类似，null 看起来也应该被当作关键字，但是同样从技术上讲，它仅仅只是空字面常量（第 3.10.7 节）。

3.2 标识符

合法标识符：

- 1、由大小写英文字母 (a-z,A-Z)，数字 (0-9)，下划线 (_)，美元符号 (\$) 组成；
- 2、不能以数字开头；
- 3、不可以使用关键字、保留字、true、false、null
- 4、标识符严格区分大小写，长度无限制
- 5、标识符不能包含空格

标识符命名风格：

- 1.【强制】代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。
- 2.【强制】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式
- 3.【强制】类名使用 UpperCamelCase 风格，但以下情形例外：DO/BO/DTO/VO/AO/PO/UID 等。
- 4.【强制】方法名、参数名、成员变量、局部变量都统一使用 lowerCamelCase 风格，必须遵从驼峰形式
- 5.【强制】常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。
- 6.【强制】抽象类命名使用 Abstract 或 Base 开头；异常类命名使用 Exception 结尾；

测试类 命名以它要测试的类的名称开始，以 Test 结尾

7. 【强制】包名统一使用小写，点分隔符之间有且仅有一个自然语义的英语单词。包名统一使用单数形式，但是类名如果有复数含义，类名可以使用复数形式。
8. 【参考】枚举类名带上 Enum 后缀，枚举成员名称需要全大写，单词间用下划线隔开。

引用自：《Java 开发手册》——阿里巴巴

类型	约束	例
项目名	全部小写，多个单词用中划线分隔 '-'	spring-cloud
包名	全部小写	com.alibaba.fastjson
类名	单词首字母大写	Feature, ParserConfig, DefaultFieldDeserializer
变量名	首字母小写，多个单词组成时，除首个单词，其他单词首字母都要大写	password, userName
常量名	全部大写，多个单词，用 '_' 分隔	CACHE_EXPIRED_TIME
方法	同变量	read(), readObject(), getByte()

图片引用自：<https://zhuanlan.zhihu.com/p/96100037>

单下划线作为标识符是不可行的，但是双下划线是可行的。

```
public class _ {  
    public static void main(String[] args) {  
        int _ = 99;  
        System.out.println(_);  
    }  
}
```

```
_. java:1: 警告: '_' 用作标识符  
public class _ {  
  
    (Java SE 8 之后的发行版中可能不支持使用 '_' 作为标识符)  
_. java:3: 警告: '_' 用作标识符  
    int _ = 99;  
  
    (Java SE 8 之后的发行版中可能不支持使用 '_' 作为标识符)  
_. java:4: 警告: '_' 用作标识符  
        System.out.println(_);  
  
    (Java SE 8 之后的发行版中可能不支持使用 '_' 作为标识符)  
3 个警告
```

4 第三天：P31-P45 运算符

4.1 算术运算符

表 4-1 算术运算符

运算符	含义	例子	结果
+	正号	+3	3
-	负号	b = 4; -b	-4
+	加	5 + 5	10
-	减	6 - 4	2
*	乘	3 * 4	12

/	除	10 / 4	2
		(double) 10 / 4	2.5
%	取余	7 % 5	2
		-3 % -2	-1
		-3 % 2	-1
		3 % -2	1
++	前置自增：先运算后取值	a = 2; b = ++a	a = 3; b = 3
++	后置自增：先取值后运算	a = 2; b = a++	a = 3; b = 2
--	前置自减：先运算后取值	a = 2; b = --a	a = 1; b = 1
--	后置自减：先取值后运算	a = 2; b = a--	a = 1; b = 2
+	字符串连接符	"he" + "llo"	"hello"

注意：

- 1、整数除法的结果是整数，即求整除的商。
- 2、取余运算只适用于两个整数，结果的正负号和被取余数的正负号相同。

4.2 赋值运算符

=、+=、-=、*=、/=、%=、<<=、>>=、&=、^=、|=

设 byte b = 10;，b += 2 并不等于 b = b + 2。后者会进行数据类型提升，而前者数据类型不发生转换。

设 int i = 1;，i *= 0.1 可以相当于 i = (int)(i * 0.1)。结果为 0。上面的 b += 2 可以相当于 b = (byte)(b + 2)。

```
public class SetValue {
    public static void main(String[] args) {
        byte b = 10;
        b += 2; // 不会进行类型转换
        System.out.println(b);

        b = b + 2; // 报错：类型不兼容
        System.out.println(b);

        int i = 1;
        i *= 0.1; // 0, 相当于 i = (int)(i * 0.1)
        System.out.println(i);
    }
}
```

【面试题】

```
short s = 3;
s = s+2; ①
s += 2; ②
①和②有什么区别？
```

答：①编译不通过，因为 short 在做运算时，会先自动类型提升为 int 类型，所以必须用

int 类型的变量来接收结果。

②编译通过，s 的值变为 5。+= 不会改变原来的数据类型，相当于 `s=(short)(s+2)`。

4.3 关系运算符（比较运算符）

表 4-2 关系运算符

运算符	含义	例子	结果
==	等于	4 == 3	false
!=	不等于	4 != 3	true
<	小于	4 < 3	false
>	大于	4 > 3	true
<=	小于等于	4 <= 3	false
>=	大于等于	4 >= 3	true

关系运算符的结果都是 boolean 类型的，值要么是 true，要么是 false。

这里说明一下，Java 和 C/C++ 不一样，在 Java 中 if 语句的小括号里面的值或者表达式的值必须为 boolean 类型，否则会报错：类型不兼容。因此如果是一条 int 类型的赋值语句，则不能作为 if 的条件（在 C/C++ 中是可以的）。但需要注意的是如果赋值表达式的左值是 boolean 类型的，那么赋值表达式的值也会是 boolean 类型的。代码如下：

```
public class Equals {
    public static void main(String[] args) {
        boolean boo = false;

        // 这里不会报错，因为赋值语句的值是表达式最左边变量的值
        // 即就是 boo 的值，是 boolean 类型的
        if(boo = true)
        {
            System.out.println("第一个 if 里面的语句执行了");
        }

        int number = 0;
        // 这里会报错：类型不兼容。因为表达式 number = 3 的值是 number 的值 3
        // 而 if 里面所需要的值必须为 boolean 类型
        if(number = 3)
        {
            System.out.println("第二个 if 里面的语句执行了");
        }

        System.out.println("完成! ");
    }
}
```

4.4 逻辑运算符

& 逻辑与 | 逻辑或 ! 逻辑非
&& 短路与 || 短路或 ^ 逻辑异或

a	b	a & b	a && b	a b	a b	!a	a ^ b
true	true	true	true	true	true	false	false
true	false	false	false	true	true	false	true
false	true	false	false	true	true	true	true
false	false	false	false	false	false	true	false

异或：异为真。不一样，则结果为真。

说明：逻辑运算的操作数都是 boolean 类型，运算结果也是 boolean 类型。

【面试题】&与&&, |与||的区别。

&和&&的区别? |和||的区别?
I
&和&&如果左边为true的时候，右边的式子都会执行。
&左边为false时，右边的式子仍会执行。
&&左边为false时，右边的式子将不再执行。（因为可以推出来结果）
|和||如果左边为false的时候，右边的式子都会执行。
|左边为true时，右边的式子仍会执行。
||左边为true时，右边的式子将不再执行。（因为可以推出来结果）

答：

```
public class Logic {
    public static void main(String[] args) {
        int x, y;
        x = 0; y = 10;
        // 先判断 x==0, 0==0 为 true, 再执行 x++, x=1;
        // 前面的为 true, 根据短路性, 后面的不执行, y=10。
        if( (x++ == 0) || ((y=20) == 20) )
        {
            System.out.println("y="+y);
        }

        // 先执行 x++, x=1; 再判断 x==0, 1==0 为 false;
        // 前面的为 false, 不能直接推断出结果, 执行后面的语句, y=20。
        if( (++x == 0) || ((y=20) == 20) )
        {
            System.out.println("y="+y);
        }

        int a, b;
        a = 0; b = 10;
        // 先判断 x==0, 0==0 为 true, 再执行 x++, x=1;
        // 前面的为 true, 没有短路性, 执行后面的语句, y=20。
        // 或者把 | 理解为位运算操作符。
    }
}
```

```

        if( (a++ == 0) | ((b=20) == 20) )
        {
            System.out.println("b="+b);
        }
    }
}

```

y=10
 y=20
 结果: b=20

这篇文章对&和|的理解思路很清奇,认真看完,不要看到第一个问题就不看了。文章地址:
<https://blog.csdn.net/websph/article/details/5669363>

4.5 位运算符

4.5.1 进制

世界上有 10 种人,一种是都二进制的,另一种是不懂二进制的。

- 二进制: 0, 1, 以 0b 或者 0B 开头;
- 十进制: 0-9;
- 八进制: 0-7, 以 0 开头;
- 十六进制: 0-9 及 A-B, 以 0x 或者 0X 开头。A-F 大小写不区分。

4.5.2 原码 反码 补码

(1) 正数的原码、反码、补码都相同。(9 的原码: 0000 1001)

(2) 负数

负数的原码: 最高位为符号位, 1 表示负数。即把其对应的正数的原码的符号位改为 1。(-9 的原码: 1000 1001)

负数的反码: 符号位不变, 把负数的原码的其它位按位取反。(-9 的反码: 1111 0110)

负数的补码: 反码加一。(-9 的补码: 1111 0111)

(byte)128 => -128

128: 0000 0000, 0000 0000, 0000 0000, 1000 0000

(byte)128: 1000 0000 => -128

4.5.3 位运算

Java 定义了位运算符, 应用于整数类型(int), 长整型(long), 短整型(short), 字符型(char), 和字节型(byte)等类型。位运算符作用在所有的位上, 并且按位运算。

运算符	含义	例子	结果
设 A = 0011 1100; B = 0000 1101			
&	按位与	A & B	12, 即 0000 1100
	按位或	A B	61, 即 0011 1101
^	按位异或	A ^ B	49, 即 0011 0001
~	按位取反	~A	- 61, 即 1100 0011
<<	按位左移	A << 2	240, 即 1111 0000
>>	按位右移(补符号位)	A >> 2	15, 即 1111
>>>	按位右移(补零)		

<< 左移：规则是带符号位移，高位移出，低位补 0，移动位数超过该类型的最大位数，则进行取模，如对 Integer 型左移 34 位，实际上只移动了两位。左移一位相当于乘以 2 的一次方，左移 n 位相当于乘以 2 的 n 次方。

>> 右移：规则是低位移出，高位补符号位，移动位数超过该类型的最大位数，则进行取模，如对 Integer 型左移 34 位，实际上只移动了两位。

>>> 无符号右移：无符号位移是什么意思呢，就是右移的时候，无论正负数，高位始终补 0。当然，它也仅仅针对负数计算有意义。

直接上代码：

```
public class Bit {
    public static void main(String[] args) {
        byte a = -1; // 1111 1111

        byte b1 = (byte)(a >> 4);
        byte b2 = (byte)(a >>> 4);

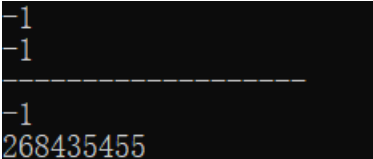
        System.out.println(b1); // -1
        System.out.println(b2); // -1
        // byte 类型运算时会转为 int 类型，后强转为 byte 后只有低 8 位，全为 1。

        System.out.println("-----");

        int i = -1; // 1111 1111

        // 高位补符号位 1111 1111 1111 1111, 1111 1111 1111 1111
        int c1 = i >> 4;
        // 高位补零 0000 1111 1111 1111, 1111 1111 1111 1111 ==
268435455
        int c2 = i >>> 4;

        System.out.println(c1);
        System.out.println(c2);
    }
}
```



```
-1
-1
-----
-1
结果: 268435455
```

这个有意思哦~ 并不是循环左移

```
System.out.println(1 << 31);
System.out.println(3 << 31);
```

```

System.out.println(3 << 32);
System.out.println(3 << 33);
System.out.println((3 << 31) << 1); // 这个好玩
System.out.println(-1 << 32);
System.out.println(-1 << 33);
System.out.println(-1 << 34); // 得-4 不是循环左移

```

结果：

```

2147483648
2147483648
3
6
0
-1
-2
-4

```

4.6 三元运算符

(条件表达式) ? 表达式 1 : 表达式 2

- 1、条件表达式的结果为 boolean 类型。
- 2、条件表达式为 true, 执行表达式 1; 为 false, 执行表达式 2。
- 3、表达式 1 和表达式 2 的类型必须相同。或者可以进行类型转换。
- 4、三元运算符可以被嵌套使用, 但不建议这样使用。
- 5、三元运算符可以被 if 语句替代; 反之, 不成立。

4.7 运算符的优先级

表 4-3 Java 运算符优先级列表

优先级	运算符	简介	结合性
1	. [] () {} ; ,		从左向右
2	! + - ~ ++ -- (DataType)	一元运算符	从右向左
3	* / %	乘、除、取模 (余数)	从左向右
4	+ -	加减法	从左向右
5	<< >> >>>	左位移、右位移、无符号右移	从左向右
6	< <= > >= instanceof	关系运算符, 对象类型判断	从左向右
7	== !=	等于, 不等于。	从左向右
8	&	按位与	从左向右
9	^	按位异或	从左向右
10		按位或	从左向右
11	&&	短路与	从左向右
12		短路或	从左向右
13	?:	条件运算符	从右向左
14	= *= /= %=	混合赋值运算符	从右向左
	+= -= <<= >>=		
	>>>= &= ^= =		

5 第四天：P46-P73 程序流程控制

5.1 顺序结构

程序从上到下逐行地执行，中间没有任何判断和跳转。

5.2 分支结构

根据条件，选择性地执行某段代码。

5.2.1 if – else

就近原则：else 匹配上面离其最近的一个 if。

1、单个 if

```
if(判断条件)
{
    /* 执行代码段 */
}
```

2、if – else

```
if(判断条件)
{
    /* 执行代码段 */
}
else
{
    /* 执行另一个代码段 */
}
```

3、if - else if - ... - else

```
if(判断条件 1)
{
    /* 执行代码段 1 */
}
else if(判断条件 2)
{
    /* 执行代码段 2 */
}
// .....
else
{
    /* 执行代码段 n */
}
```

有多个条件表达式：如果关系是互斥的，那么顺序无所谓；

如果关系是包含的，那么范围小的在上面，范围大的在下面。

5.2.2 switch – case

```
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;
}
```

- 1、程序执行到 break 才退出 switch 语句块；若没有写 break 语句，则继续执行（case 穿透）。
- 2、表达式的值必须是 byte、short、char、int、枚举、String 类型的。
- 3、case 后只能是常量，且不能有重复的值。
- 4、default 是可选的，其位置是灵活的，只有当没有匹配到一条 case 时才执行。
- 5、先对所有的 case 进行匹配，如果匹配到了就不会执行 default 语句。

可以省略 break 来做一些有趣的事情：如下面，常量 A 匹配时要做一些事情；常量 B 匹配时要比匹配 A 时多做一些事情，且也要做 A 做的事情。

```
switch(表达式)
{
    case 常量 B:
        做 B 要多做的事情;
        // 没有 break 语句，程序继续执行 称作 case 穿透
    case 常量 A:
        A 要做的事情;
        break;
    default:
        语句;
        break;
}
```

5.3 循环结构

根据循环条件，重复性地执行某段代码。

- 1、初始化语句
- 2、循环条件

- 3、循环体
- 4、迭代语句

5.3.1 for

```
for(初始化语句; 循环条件; 迭代语句)
{
    循环体;
}
```

执行顺序: 1 → 2 → 3 → 4 → 2 → 3 → 4 → → 2 → 3 → 4 → 2 → 退出

循环体执行完成后执行迭代语句; 退出则是从循环条件处退出。

确认一下 break 后不执行迭代语句, 而 continue 后执行迭代语句:

```
public class Loop {
    public static void main(String[] args) {
        int i;

        for(i = 0; i < 3; i++) {
            System.out.println(i);
            continue; // continue 后执行 i++
        }
        System.out.println(i);

        System.out.println("-----");
        for(i = 0; i < 3; i++) {
            System.out.println(i);
            if(i == 2)
                break; // break 后不执行 i++
        }
        System.out.println(i);
    }
}
```

}结果: 

5.3.2 while

```
初始化语句;
while(循环条件)
{
    循环体;
```

```
    迭代语句;
}
```

执行顺序: 1 → 2 → 3 → 4 → 2 → 3 → 4 → → 2 → 3 → 4 → 2 → 退出

5.3.3 do – while

```
初始化语句;
do
{
    循环体;
    迭代语句;
} while(循环条件);
```

执行顺序: 1 → 3 → 4 → 2 → 3 → 4 → 2 → 3 → 4 → → 2 → 3 → 4 → 2 → 退出

```
do-while,while和for的区别?

do-while和while的区别?
    while可能一次循环体都不执行, 但是do-while至少执行一次循环体。

while和for的区别?
    while的初始化条件在结构外, for循环的初始化条件在结构内 (也可以在结构外)。
    while和for可以相互替换使用。
```

提一嘴: 字符串比较用 equals 方法:

6. **【强制】** Object 的 equals 方法容易抛空指针异常, 应使用常量或确定有值的对象来调用 equals。

正例: "test".equals(object);

反例: object.equals("test");

说明: 推荐使用 java.util.Objects#equals (JDK7 引入的工具类)。

打个九九乘法表呗:

```

public class MultiplicationTable {
    public static void main(String[] args) {
        for(int i = 1; i <= 9; i++)
        {
            for(int j = 1; j <= i; j++)
            {
                System.out.print(j + "*" + i + "=" + j*i + "\t");
            }
            System.out.println();
        }
    }
}

```

```

1*1=1
1*2=2   2*2=4
1*3=3   2*3=6   3*3=9
1*4=4   2*4=8   3*4=12  4*4=16
1*5=5   2*5=10  3*5=15  4*5=20  5*5=25
1*6=6   2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7   2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8   2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9   2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81
请按任意键继续. . .

```

那再来个质数吧：

```

public class PrimeNumber {
    public static void main(String[] args) {

        long startTime = System.currentTimeMillis();

        boolean isPrimeNumber;
        System.out.print(2 + " ");
        for(int i = 3 ;i <= 100000; i+=2) {
            // 判断 i 是否是质数
            isPrimeNumber = true;
            for(int j = 2; j*j <= i; j++) {
                // 不是质数
                if(i % j == 0) {
                    isPrimeNumber = false;
                    break;
                }
            }
            // 输出
            if(isPrimeNumber) {
                System.out.print(i + " ");
            }
        }
    }
}

```



```

        long endTime = System.currentTimeMillis();
        System.out.print("\n" + (endTime-startTime));
    }
}
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
请按任意键继续. . .

```

5.4 continue 和 break

break :
 使用范围：循环结构，switch-case结构
 作用：1.在循环结构中用于结束当前循环
 2.在switch-case中用于结束当前分支（跳出switch-case结构）
 在嵌套循环中，结束掉的是包含它的那个循环的当前循环。

continue :
 使用范围：循环结构
 作用：用于结束当次循环

break 结束外层循环：

```

public class Break {
    public static void main(String[] args) {
        A: for(int i = 0 ; i < 10; i++) {
            for(int j = 0; j < 10; j++) {
                System.out.println("j="+j);
                if(j == 2) {
                    break A;
                }
            }
            System.out.println("i="+i);
        }
    }
}

```

```

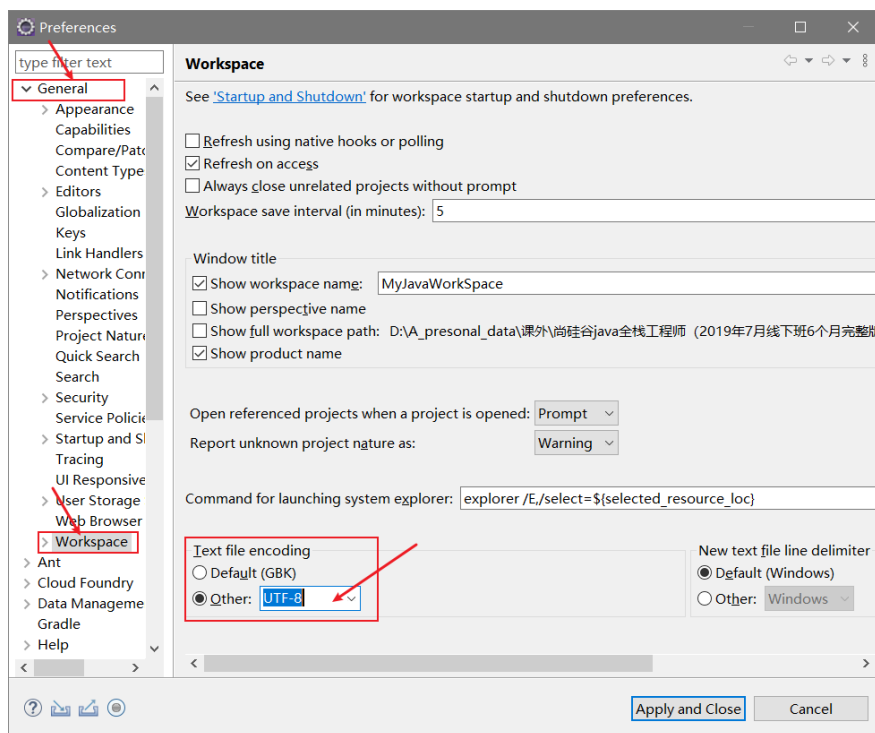
j=0
j=1
j=2
请按任意键继续. . .

```

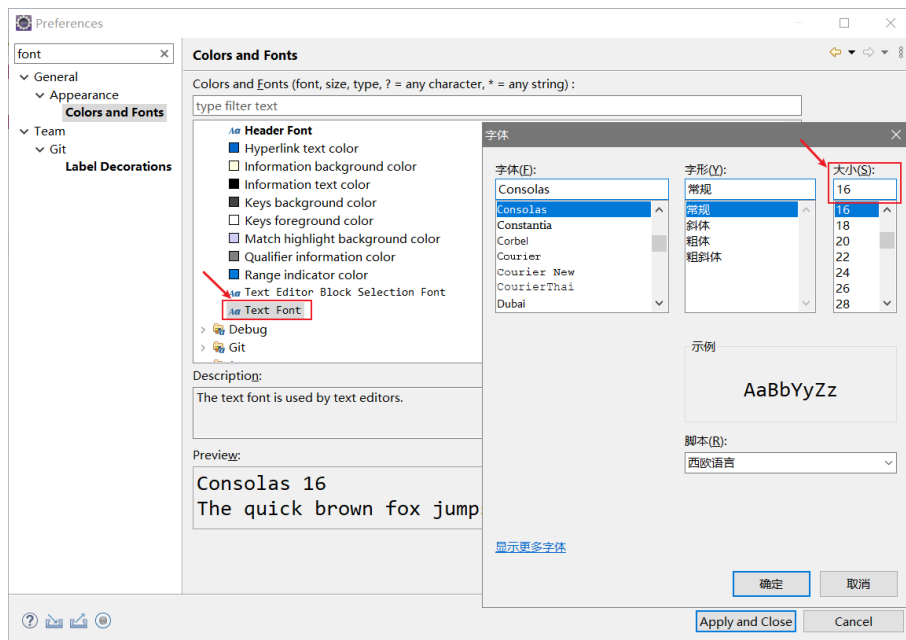
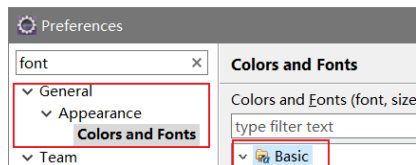
6 第五天：P74-P85 使用 Eclipse 做项目

6.1 Eclipse 配置（以后修改配置都到这里更新）

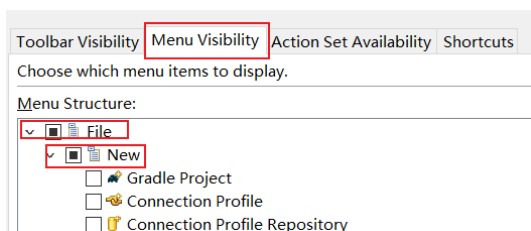
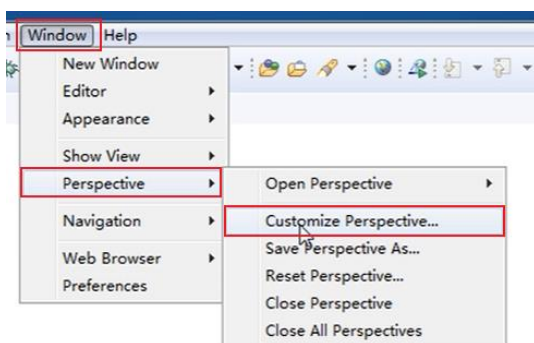
- 改为 utf-8 编码格式：Window → Preferences → General → Workspace



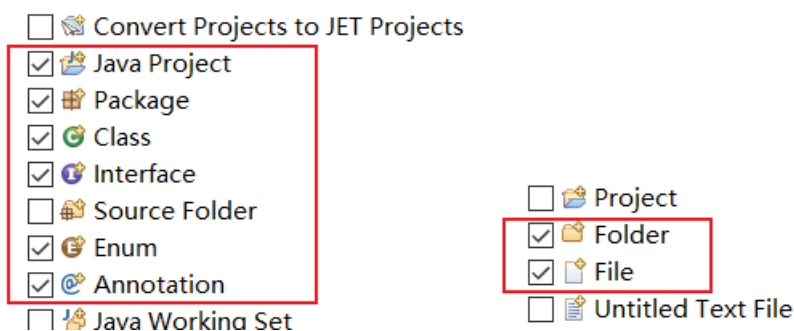
● 字体大小修改为 16 号：Window → Preferences → General → Appearance



● new 菜单的子项目修改：



要下面这些:



- 按住 Ctrl 键单击左键可以看到源代码:

点击 Attach Source 按钮, 弹出视图, 把 JDK 下的 src.zip 文件传入点击 OK 即可。

https://zhidao.baidu.com/question/2270685638380167708.html?qbl=relate_question_0

6.2 Eclipse 的小技巧

- Eclipse 中一个项目中可以有多个 main, 可以同时运行多个程序。(控制台可以切换)
- 一些快捷键
 - Alt + /: 代码提示和自动补全:
输入 main, 然后按 “Alt + /” → public static void main(String[] args)
输入 sout, 然后按 “Alt + /” → System.out.println();
 - Ctrl + D: 删除当前行。
 - Ctrl + Alt + Down: 复制当前这一行代码到下一行。
 - Ctrl + /: 单行注释与取消; Ctrl + Shift + /: 块注释; Ctrl + Shift + \: 取消块注释。
 - Ctrl + Shift + T: 打开 Open Type 查找类文件。
 - Ctrl + T: 查看类的继承关系。
 - Ctrl + I: 等号左边返回值类型代码自动补全。

7 第六天: P86-P97 数组

数组: 相同数据类型的变量的集合。

- 1、在 Java 中数组本身属于引用数据类型。

-
- 2、数组中的元素既可以是基本数据类型，也可以是引用数据类型。
 - 3、**Java 的数组都是分配在堆上的。**
 - 4、数组创建完成后，长度不可以改变。

数组的声明和初始化：

```
// 数组的声明
String[] name;
int cs[]; // 不建议这种的
// 静态初始化
name = new String[]{"1", "2", "3", "4"};
int numbers[] = {1, 2, 3, 4};
// 动态初始化
cs = new int[5];
double d[] = new double[10];
```

1. 使用代码描述一维和二维数组的声明的初始化。

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

int[] numbers = new int[5];

int[][] ns = new int[][] {{1, 2}, {2, 3}};
int[][] ns = {{1, 2}, {2, 3}};

int[][] ns = new int[2][3];
int[][] ns = new int[2][];
ns[0] = new int[5];
ns[1] = new int[2];
```

数组元素的默认值：

- byte、short、int、long → 0
- float、double → 0.0
- boolean → false
- char → \u0000
- 引用类型 → null

数组的属性 length 用来表示数组长度。

数组的下标从 0 开始到 (length - 1) 结束。

```
int[] arr = new int[10];
for (int i = 0; i < arr.length; i++) {
```

```
    arr[i] = i;
}
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

数组下角标异常: `ArrayIndexOutOfBoundsException`

空指针异常: `NullPointerException`

`Arrays` 类的使用:

```
int[] numbers = {2,1,89,21,45,-1};

//排序 sort(int[] numbers) 对数组中的元素进行排序
Arrays.sort(numbers);

//如果只是为了查看数组中的元素的内容,使用下面的方式更方便
//toString(int[] numbers)将数组中的所有元素以字符串的形式进行返回
String str = Arrays.toString(numbers);
System.out.println(str);

//二分法查找(前提先排序 - 查找到的位置是排序后的位置)如果没有排序,那么查找到的位置可能是错误的
//如果查找不到返回负数
int index = Arrays.binarySearch(numbers, 33);
System.out.println(index);

//fill(int[] numbers, int number)将原数组中所有的内容变成 20
Arrays.fill(numbers, 20);
System.out.println(Arrays.toString(numbers));
```

8 第七天: P98-P142 面向对象编程基础

类是对象的抽象,对象是类的实例。

类的成员: 属性、方法、构造器、代码块、内部类。

面向对象三大特征: 封装性、继承性、多态性。

面向过程强调的是功能行为;面向对象强调的是具备了功能的类。

匿名对象: `new Object()`,一般用于方法的实参。

8.1 包 (package 与 import)

包的作用: 为了对 Java 类进行统一的管理。

包命名规范: 包名统一使用小写,点分隔符之间有且仅有一个自然语义的英文单词或者多个单词自然连接到一块(如 `springframework`, `deepspace` 不需要使用任何分割)。包名统一使用单数形式,如果类名有复数含义,则可以使用复数形式。包名的构成可以分为以下几部分【前缀】【发起者名】【项目名】【模块名】。常见的前缀可以分为以下几种:

前缀名	例	含义
indi (或 onem)	indi.发起者名.项目名称.模块名.....	个体项目, 指个人发起, 但非自己独自完成的项目, 可公开或私有项目, copyright主要属于发起者。
pers	pers.个人名.项目名称.模块名.....	个人项目, 指个人发起, 独自完成, 可分享的项目, copyright主要属于个人
priv	priv.个人名.项目名称.模块名.....	私有项目, 指个人发起, 独自完成, 非公开的私人使用的项目, copyright属于个人。
team	team.团队名.项目名称.模块名.....	团队项目, 指由团队发起, 并由该团队开发的项目, copyright属于该团队所有
顶级域名	com.公司名.项目名称.模块名.....	公司项目, copyright由项目发起的公司所有

每个“.”代表一层目录。

不同的包下面可以有相同的类名, 同一个包下不能有相同的类名。(虽然 Java 严格区分大小写, 同一个包下可以有 DbContent 和 dbcontent; 但是文件系统不区分大小写, 导致文件名重复: Type with same name but different case exists.【具有相同名称但大小写不同的类型。】)

package 位于源文件的首行, 即 package 语句是代码的第一行非空行。

import:

- 1、位于 package 语句后面, 类的定义前面。
- 2、用于显示地导入指定包下的类或者接口。
- 3、如果需要导入多个类或者接口, 则需要并列地写多条 import 语句。
- 4、也可以使用“import java.util.*;”这种方式导入该包下的所有类或接口。
 - 1、但是不会导入子包, 例如: import java.util.*; 可以导入 Scanner 类, 而 import java.*;不能。
 - 2、这种方法并不会导入所有的类, 只会导入用到的类, 编译时会替换成单类型导入。
- 5、已经导入了一个包, 如果还需要它的子包, 那么子包还需另外导入。
- 6、java.lang 包下的类和接口是默认自动导入的, 当前包的成员本身就在作用域内所以当前包和 java.lang 包的导入是可以省略的。
- 7、如果有使用到不同包下的同名类, 只能使用类的全名来指明。
- 8、import static 调用指定包下指定类或接口中的静态属性和方法。

```
import static java.lang.System.out;
out.println("hello");
```

参考: https://blog.csdn.net/qj_25665807/article/details/74747868

8.2 访问权限修饰符

对于 class 的访问权限只能用 public 和 default (缺省) 来修饰:

public: 任何地方可以访问该类;

default: 只有在同一个包下访问该类。

代码块只能用 static 修饰。

表 8-1 类的成员(属性、方法、构造器、内部类) 的访问权限

修饰符	类内部	同一个包	不同包的子类	任何地方
private	√			
(缺省)	√	√		
protected	√	√	√	
public	√	√	√	√

8.3 this

this 表示当前对象。可以用来使用类的属性、方法和构造器。

this.属性名：用来指定当前对象的属性，this 在不引起歧义时可以省略。

this.方法名()：用来调用当前对象的非静态方法，this 在不引起歧义时可以省略。

this()用于调用类的其他构造器。可以多级调用，但禁止套娃。this()必须放在构造器的首行，一个构造器中只能出现一个 this()。→ 推论：n 个构造器中一共最多可出现 n-1 条 this() 语句。

this 不能出现在静态上下文中。

8.4 super

super 表示当前对象的父类。可以用来使用父类的属性、方法和构造器。

当父类中的方法被子类重写后，在子类中可以使用【super.方法名】来调用父类的被重写方法，使用【this.方法名】来调用子类的重写方法。如果子类没有重写，那么【super.方法名】和【this.方法名】都是调用父类的方法，或者直接使用【方法名】（省略了 this.）。属性同理。

如果有一个方法继承时一路重写下来，那么【super.方法名】调用的是直接父类的方法。

super()用来调用父类的构造器。在子类的构造器中，如果没有显示地使用 super()调用父类构造器，或者使用 this()来调用其他构造器，那么 Java 默认自动的调用父类的无参构造器。如果父类没有无参构造器，就会报错【Implicit super constructor Person() is undefined. Must explicitly invoke another constructor】。（这也就解释了为什么 Java 会白送一个无参构造器）

super()必须放在子类构造器的首行，一个构造器中只能出现一个 super()。

创建子类对象必定会调用父类构造器。

No.	区别点	this	super
1	访问属性	访问本类中的属性， 如果本类没有此属性 则从父类中继续查找	访问父类中的属性
2	调用方法	访问本类中的方法	直接访问父类中的方法
3	调用构造器	调用本类构造器，必须放在构造器的首行	调用父类构造器，必须放在子类构造器的首行
4	特殊	表示当前对象	子类中访问父类对象

8.5 static

static 可以用来修饰：属性、方法、代码块、内部类。

8.5.1 static 修饰变量：类变量

1、同一个类的所有对象共同拥有(共享)一份类变量；每个对象各自拥有一份实例变量。

- 2、类变量随着类的加载而创建；实例变量随着对象的创建而创建。
- 3、在程序的一次运行过程中，每个类只会被加载一次。
- 4、使用类变量：类名.类变量名 / 对象名.类变量名。

8.5.2 static 修饰方法：静态方法

- 1、静态方法随着类的加载而加载。
- 2、调用静态方法：类名.静态方法名(参数列表) / 对象名.静态方法名(参数列表)。
- 3、静态方法中不能使用非静态的实例变量，可以使用静态的类变量。静态方法中不能调用非静态的方法，可以调用静态方法。
- 4、非静态方法中可以使用类变量和静态方法。
- 5、静态方法中不能使用“this”和“super”。

思考：什么时候使用 static 修饰属性和方法？

static 修饰属性：

- 1、当一个属性作为常量时，必须使用 static 修饰。
- 2、当多个对象共同使用一份属性时。

static 修饰方法：

- 1、工具类中的方法一般使用 static 修饰。
- 2、有时为了使用类变量，方法也会使用 static 修饰。

8.5.3 类加载的过程

- 1、当我们创建对象时，首先会在方法区中查找该类信息。
- 2、如果在方法区中没有找到该类信息，则进行类加载。如果有，直接创建对象。
- 3、类加载：将字节码文件加载到 JVM 中，同时在方法区的特定区域中存放 static 变量。
(细节：不是只有创建对象时才进行类加载，而是当我们调用类时就进行类加载)

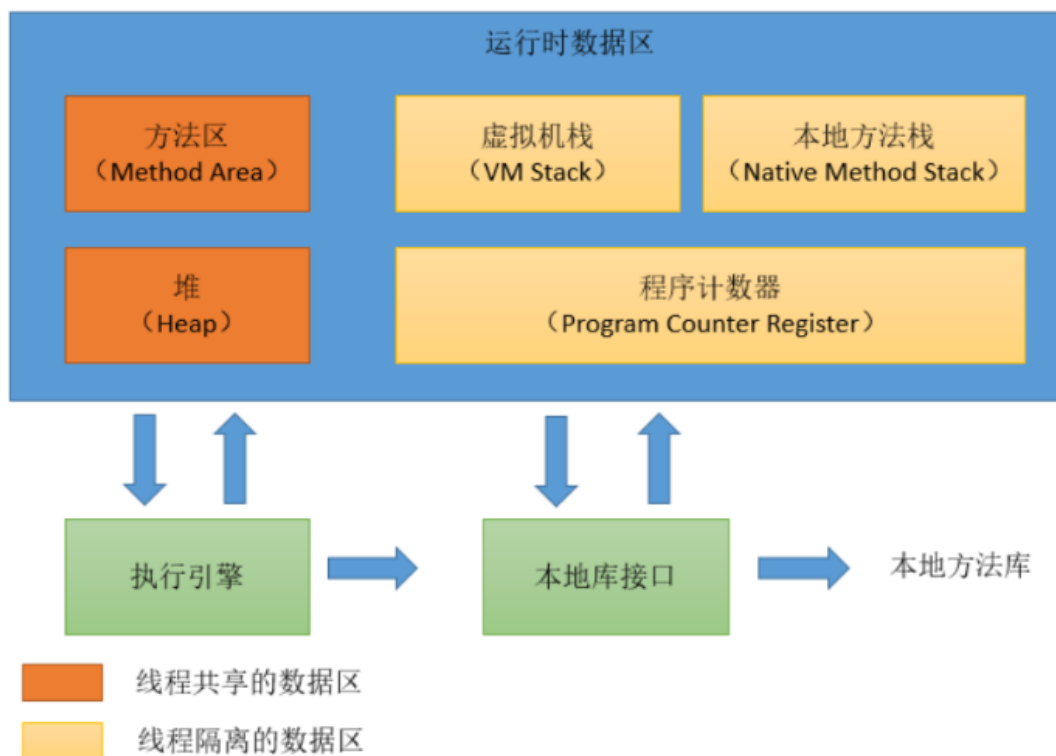


图 8-1 JVM 内存区域

8.5.4 单例(Singleton)设计模式

设计模式：设计模式是在大量的实践中总结和理论化之后优选的代码结构、编程风格、以及解决问题的思考方式
常见的有23种设计模式：单例设计模式，代理设计模式，装饰设计模式，观察者设计模式.....

单例 (Singleton)设计模式

设计模式是在大量的实践中总结和理论化之后优选的代码结构、编程风格、以及**解决问题的思考方式**。设计模式就像是经典的棋谱，不同的棋局，我们用不同的棋谱，免去我们自己再思考和摸索。”套路”

所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将类的**构造方法的访问权限设置为private**，这样，就不能用new操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能**调用该类的某个静态方法**以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的**该类对象的变量也必须定义成静态的**。

让天下没有难学的技术

单例设计模式：在一个项目中某个类自始至终只有一个实例化的对象。

单例模式主要有两种：饿汉式、懒汉式。其中饿汉式是线程安全的；懒汉式是线程不安全的，但是延迟了对象创建时机，一定程度上节省了内存开销。

```
/* 饿汉式 */
class Singleton {
    // 1.私有化构造器
    private Singleton () {}
    // 2.类内部创建一个本类的对象
    private static Singleton t = new Singleton ();
    // 3.提供一个方法来获取实例对象
    public static Singleton getInstance() {
        return t;
    }
}
```

```
/* 懒汉式 */
class Singleton {
    // 1.私有化构造器
    private Singleton () {}
    // 2.类内部创建一个本类的对象
    private static Singleton t = null;
    // 3.提供一个方法来获取实例对象
    public static Singleton getInstance() {
```

```

        if(t == null) {
            t = new Singleton();
        }
        return t;
    }
}

```

懒汉式线程不安全测试

```

public class SingletonTest{
    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            new Thread(
                () -> {
                    System.out.println(Singleton.getInstance());
                }
            ).start();
        }
    }
}

```

```

priv.song.instance.Singleton@6423eb7d
priv.song.instance.Singleton@6423eb7d
priv.song.instance.Singleton@6423eb7d
priv.song.instance.Singleton@6423eb7d
priv.song.instance.Singleton@6423eb7d
priv.song.instance.Singleton@6423eb7d
priv.song.instance.Singleton@1dbdc081
priv.song.instance.Singleton@6423eb7d

```

解决方法：在 getInstance 方法上加同步

```

public static synchronized Singleton getInstance() {
    if(t == null) {
        t = new Singleton();
    }
    return t;
}

```

好好看看这篇文章：<https://blog.csdn.net/jason0539/article/details/23297037>

8.6 final

final 修饰的类：不能被继承。如，String、StringBuffer。

final 修饰的方法：不能被重写。

final 修饰的属性：必须要且仅能被赋值一次(即初始化)，且只有显示赋值、代码块赋值、构造器赋值这三种方式。

final 修饰的方法体/代码块的局部变量：必须要且仅能被赋值一次，可以声明的同时赋

值，也可以先声明后赋值。

`final` 修饰的方法的形参，保证参数不被改变（如果参数是引用类型，只保证引用不改变所指对象，而对象的属性的值可以改变）。

常量：可以通过类名调用，`static`；常量的值不可改变，`final`；名称全大写。

```
public static final double PI = 3.14159265358979323846;
```

8.7 JavaBean

JavaBean 是一种 Java 语言写成的可重用组件。

所谓 JavaBean，是指符合以下标准的 Java 类：

- 类是具体的，且是 `public` 的；
- 有一个无参数的 `public` 的构造器；
- 有属性，且每一个属性具有对应的 `get`、`set` 方法。

9 第七天：P98-P142 类的成员

9.1 属性 Field

局部变量和成员变量的相同点和不同点：

相同点：

- 1、都是变量，都要先声明后使用
- 2、都可以被多次赋值
- 3、都有其作用域
- 4、声明方式：变量类型 变量名

不同点：

- 1、声明位置：

成员变量声明在类内部，方法、构造器等结构外部；

局部变量声明在方法中、方法的形参、构造器中、构造器的形成等。

- 2、默认值

成员变量默认值：

byte、short、int、long → 0

float、double → 0.0

boolean → false

char → '\u0000'

引用类型 → null；

局部变量没有默认值。

- 3、权限修饰符

成员变量可以被权限修饰符修饰；

局部变量不能。

- 4、内存

成员变量存储在堆上；

局部变量存储在栈上。

9.2 方法 Method

权限修饰符 返回值类型 方法名(参数列表) { 方法体

```
}
```

- 1、方法可以被权限修饰符所修饰。
- 2、使用 return 语句来返回一个数据；在 return 语句处结束方法，返回上一层调用。
- 3、调用方法：对象名/类名.方法名(实参列表)。
- 4、形参列表中多个参数用逗号（,）分开。

9.2.1 方法的参数传递：值传递

形式参数：方法声明时参数列表中的参数。

实际参数：方法调用时实际传递给方法的参数。

在 Java 中，方法的参数传递方式只有一种：值传递。即将实参的值复制一份传递到方法内，实参本身不受影响。按值传递重要特点：传递的是值的拷贝，也就是说传递后就互不相关了。

对于基本类型来说，传递的是实参的副本（值传递），故在方法内修改传递进来的值并不会影响实参本身；

对于引用类型来说，传递进来的是引用变量的副本（也是值传递），因此该副本与实参均是引用变量，他们均可以操作所引用的对象，在方法内通过引用变量对堆区的对象进行操作时均会对该对象有影响。由于传入方法的是实际参数值的副本，因此，参数本身不会受到任何影响！

9.2.2 方法重载 overload

9.2.2.1 方法重载的定义

方法重载：同一个类中，方法名相同，形参列表不同，则构成方法重载。

不同的形参列表指：形参类型不同，个数不同，顺序不同。

注意：

- 1、只有返回值不同，不是重载，会报错；
- 2、只有形参的变量名不同，不是重载，会报错；
- 3、只有权限修饰符不同，不是重载，会报错。

9.2.2.2 方法重载时的调用选择

```
public class MethodTest {
    public static void main(String[] args) {
        int a = 0;
        method(a);
    }

    // 以下五个方法按照调用选择顺序排列
    //方法 1
    public static void method(int a){
        System.out.println("执行 method(int a)");
    }
    //方法 2
    public static void method(long a){
        System.out.println("执行 method(long a)");
    }
    //方法 3
```

```
public static void method(Integer a) {
    System.out.println("执行 method(Integer a)");
}
//方法 4
public static void method(Object a) {
    System.out.println("执行 method(Object a)");
}
//方法 5
public static void method(int...a) {
    System.out.println("执行 method(int...a)");
}
}
```

```
class A {
    public static void method(int a){
        System.out.println("A 类中的 method()");
    }
}
public class MethodTest2 extends A{
    //方法 4
    public static void method(Object a) {
        System.out.println("执行 method(Object a)");
    }
    //方法 5
    public static void method(int...a) {
        System.out.println("执行 method(int...a)");
    }
    public static void main(String[] args) {
        int a = 0;
        method(a);
    }
}
```

关于方法重载时的调用选择我们可以得出以下结论：

- (1) 精确匹配：对于上述代码中，当有 `method(int a)` 存在时调用的肯定就是这个方法；
- (2) 自动类型提升：对于基础数据类型，自动转成表示范围更大的类型；当方法 1 被注释的时候，会去调用 `method(long a)` 而不是 `method(Integer a)`；
- (3) 自动装箱与拆箱：当方法 1, 2 被注释，就调用方法 3；
- (4) 根据子类依次向上继承路线匹配：当只有方法 4 与方法 5 时，先找 `int` 的父类，找到的 `object` 类型，匹配之后调用；当继承 `A` 类之后，由于本类没有合适的方法，然后就去 `A` 类中找，匹配调用 (`A` 类中方法参数类型换成 `long`, `Integer` 结果也一样)；
- (5) 根据可变参数匹配。

引用自：<https://www.jianshu.com/p/306c4bfe3f54>

9.2.3 方法的重写 override

定义:在子类中可以根据需要对从父类中继承而来的方法进行改写,也称为方法的重置、覆盖。在程序执行时,子类的方法将覆盖掉父类的方法。

要求:

- 1、重写方法必须和被重写方法具有相同的方法名、形参列表。
- 2、重写方法不能使用比被重写方法更严格的访问权限。
- 3、重写方法的返回值类型要么和被重写方法一致,要么是被重写方法返回值类型的子类。如果是基本数据类型或者 void,那么两者必须一样。
- 4、重写方法和被重写方法必须同时为非 static 的。如果同时为 static 的,不是方法重写。父类的静态方法可以被子类继承,但是不能重写,静态方法属于静态绑定。
- 5、重写方法抛出的异常不能大于被重写方法的异常。
- 6、父类中被 private 修饰的方法不可以被子类重写。如果子类中有相同的方法,则会被视作方法重载。

可以使用【@Override】注解来说明当前方法是一个重写方法。

Java 中其实没有虚函数的概念,它的普通函数就相当于 C++ 的虚函数,动态绑定是 Java 的默认行为。如果 Java 中不希望某个函数具有虚函数特性,可以加上 final 关键字变成非虚函数。

9.2.4 方法重载与方法重写的区别

【方法重载】:

- 1、同一个类中
- 2、方法名相同,参数列表不同(参数顺序、个数、类型)
- 3、方法返回值、访问修饰符任意
- 4、与方法的参数名无关

【方法重写】:

- 1、有继承关系的子类中
- 2、方法名相同,参数列表相同(参数顺序、个数、类型),方法返回值相同
- 3、访问修饰符,访问范围需要大于等于父类的访问范围
- 4、与方法的参数名无关

表 9-1 方法重载与方法重写的区别

区别点	重载方法	重写方法
参数列表	必须修改	一定不能修改
返回类型	可以修改	一定不能修改
异常	可以修改	可以减少或删除,一定不能抛出新的或者更广的异常
访问	可以修改	一定不能做更严格的限制(可以降低限制)

9.2.5 可变形参

- 1、格式,对于方法的形参,数据类型... 可变形参的参数名。
- 2、可变形参方法与同名的方法构成重载,与形参是数组的方法不构成重载(会报错)。例如: int sum(int[] array)与 int sum(int... array)只能存在一个。
- 3、可变参数在调用时传入的实参个数从 0 开始,到无穷都可以。可以有 sum()。
- 4、可变形参与数组的使用是一样的,可变形参的底层就是一个数组。
- 5、可变形参只能声明在方法形参列表的最后,且只能有一个可变形参。

9.3 构造器 Constructor

权限修饰符 类名(形参列表) { 方法体 }

作用：1、创建对象；2、给对象进行初始化。

特点：

- 1、构造器的名字和类名必须一致；
- 2、构造器不具有返回值；
- 3、不能被 static、final、synchronization、abstract、native 修饰；
- 4、可以有 return 语句，但 return 不能返回任何值。会认为只是一个与构造器同名的方法罢了

说明：

- 1、构造器只能在创建对象时，在 new 语句中被调用一次。不能再其他地方显示调用。但可以再构造器中使用 this()调用其他构造器，使用 super()调用父类构造器。
- 2、如果类中没有定义构造器，那么 Java 免费送一个访问权限为缺省的无参数的构造器；如果定义了构造器，就不会送了。
- 3、构造器可以重载。
- 4、构造器不会被继承给子类。

属性赋值:默认值 → 显示赋值(声明时赋值)/代码块 → 构造器 → set 方法或赋值语句。
显示赋值和代码块赋值属于同级的，按照由上到下的顺序执行(见图 9-1)。

构造的过程：

- 1、分配对象空间，并将对象中成员进行默认值初始化。
- 2、执行属性值的显式初始化（这里有一点变化，一会解释，但大体是这样的）。
- 3、执行构造器的方法体（先执行父类构造器，再执行子类构造器）。
- 4、将变量关联到堆中的对象上。

关于构造器，好好看看这个：<https://blog.csdn.net/yu422560654/article/details/7399566>

子类对象的实例化过程



思考:

1. 为什么`super(...)`和`this(...)`调用语句不能同时在一个构造器中出现?
2. 为什么`super(...)`或`this(...)`调用语句只能作为构造器中的第一句出现?

```
/*
 * 子类实例化过程:
 *
 * 1. 从结果上看: 子类继承父类以后, 就拥有了父类中的属性和方法。那么通过子类的对象就可以调用父类中的属性和方法。
 *
 * 2. 从过程上: 当我们创建子类对象时, 一定会调用父类, 直接父类, 间接父类.....Object的构造器
 *    当调用这些类的构造器时, JVM便会将这些类的信息进行加载。所以我们可以调用到父类中的属性和方法。
 *
 * 注意: 自始至终我们只认为创建了一个对象 - 子类对象
 */
```

9.4 代码块

● 初始化块(代码块)作用:

➤ 对Java类或对象进行初始化

● 程序中成员变量赋值的执行顺序:

声明成员变量的默认初始化



显式初始化、多个初始化块依次被执行 (同级别下按先后顺序执行)



构造器再对成员进行初始化操作



通过"对象.属性"或"对象.方法"的方式, 可多次给属性赋值

图 9-1 属性赋值的顺序

格式：{ }

代码块只能被 `static` 所修饰。

代码块分为静态代码块和非静态代码块。

静态代码块：

- 1、用来对类中的信息进行初始化。
- 2、静态代码块随着类的加载而执行，只执行一次。
- 3、静态代码块的执行优先于非静态代码块，因为要先加载类，后创建对象。
- 4、多个静态代码块按照从上到下的顺序依次执行。
- 5、静态代码块不能使用非静态的属性和方法，只能使用静态的属性和方法。

非静态代码块：

- 1、用来对对象进行初始化。
- 2、非静态代码块随着对象的创建而执行，每次创建对象都会执行一次。
- 3、非静态代码块的执行优先于构造器。
- 4、多个非静态代码块按照从上到下的顺序依次执行。

其实还有两种代码块：普通代码块，类的方法的方法体；同步代码块，使用 `synchronized(){}` 包裹起来的代码块。

这篇文章 <https://blog.csdn.net/u012804721/article/details/52439311>

9.5 内部类

在一个类 A 的类体中，声明了另一个类 B。那么类 B 叫做内部类，类 A 叫做外部类。

```
// 外部类
class A {
    // 非静态成员内部类
    class B {
    }
    // 静态成员内部类
    static class C {
    }

    public void say() {
        // 局部内部类
        class D {
        }
    }
}
```

内部类作为类等成员：

- 1、内部类可以被四种权限修饰符所修饰。
- 2、内部类可以被 `static` 所修饰。
- 3、内部类可以调用外部类的属性和方法。
- 4、具有类的一切功能。

(1) 如何创建内部类对象？

- 创建非静态内部类对象：`new 外部类名().new 内部类名()`

`A.B b = new A().new B();`

或者

```
import priv.song.inner.A.B;
```

```
B b = new A().new B();
```

- 创建静态内部类对象: `new 外部类名.内部类名()`

```
A.C c = new A.C();
```

(2) 内部类如何调用外部类的结构（属性和方法）？

- 属性: 属性名(无冲突时), 外部类名.this.属性名(有冲突时)

```
name 或者 A.this.name
```

- 方法: 方法名(无冲突时), 外部类名.this.方法名(有冲突时)

```
func() 或者 A.this.func()
```

- 静态内部类不能调用非静态的属性和方法; 只能调用静态的属性和方法。调用时不要 `this` 关键字即可。同样如果非静态内部类调用静态的属性或方法也不要 `this` 关键字。

(3) 如何使用局部内部类（很少）？

局部内部类不能用 `static`, 无论方法是否是 `static` 的。

获取局部内部类对象: 巧用接口或父类, 但是不能向下转型

```
public class GetInner {  
    public static void main(String[] args) {  
        I i = getInnerClassObject();  
        i.test();  
    }  
  
    public static I getInnerClassObject() {  
        // 局部内部类  
        class D implements I {  
            @Override  
            public void test() {  
                System.out.println("D test");  
            }  
        }  
        D d = new D();  
        return d;  
    }  
}  
  
interface I {  
    void test();  
}
```

10 第八天: P126-P169 面向对象三大特征: 封装 继承 多态

10.1 封装性

为什么要使用封装性: 在创建对象后, 可以通过对象名.属性名的方式给属性赋值, 直接赋值的话, 只有变量类型和范围的约束。但是在实际场景中, 往往会有其他的约束。所以我

们采用如下方法进行限制：1、对属性的访问权限进行限制，这样可以防止属性在类的外部被使用；2、创建对应的 set 方法，通过 set 方法来给属性赋值，在方法中对属性的值进行额外的限制。

封装性的体现（狭义上）：1、属性私有化；2、使用 set/get 方法对属性赋值和取值。

封装性的体现（广义上）：1、权限修饰符；2、类可以被 public 和（缺省）修饰；3、四种权限修饰符可以修饰类的属性、方法、构造器、内部类。

良好的封装的优点：1. 良好的封装能够减少耦合。2. 类内部的结构可以自由修改。3. 可以对成员变量进行更精确的控制。4. 隐藏信息，实现细节。

10.2 继承性

```
class SubClass extends SuperClass
    // SubClass: 子类; SuperClass: 超类、父类、基类
```

- 1、子类继承父类以后，就拥有了父类的所有属性和方法（但能不能访问就不一定了）。
- 2、子类不会继承父类的构造器。
- 3、子类继承父类一定要满足“is a”的关系，即子类是一个父类。
- 4、父类中 private 的属性和方法不能被子类直接访问，但是可以有 get/set 方法使用。
- 5、父类的概念是相对的，父类分为直接父类和间接父类。
- 6、Java 中所有的类都继承 Object 类。如果一个类没有显示地继承某个类，那么这个类会默认继承 Object 类。
- 7、Java 中只有单继承，一个类只能继承一个父类。

继承的好处：1、减少了代码的冗余，提高了代码的复用性；2、提高了代码的扩展性；3、为多态提供了前提条件。

父类与子类的关系都基于一句话：子类必须具有父类的一切功能，子类可以用来代替父类。

里氏代换原则：任何基类可以出现的地方，子类一定可以出现。继承的含义是不修改但扩展。只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用。

当一个子类的实例应该能够替换任何其超类的实例时，它们之间才具有 is-A 关系。

上面这段话可以用来解释方法重写的要求【9.2.3】

可以看看这个讨论：<https://group.cnblogs.com/topic/5960.html>

10.3 多态性

多态性：一类事物的多种形态。

多态性的理解（广义上）：方法的重载、重写；子类对象的多态性。

多态性的理解（狭义上）：子类对象的多态性。

子类对象的多态性：父类的引用指向子类的对象。（参考游戏开发大作业：NPC 对象容器）

虚拟方法的调用（动态绑定）：编译看父类，运行看子类。

多态性的前提：继承与方法重写。

● 在多态的情况下，如何调用子类中的独有的方法？

答：向下转型。

```
Person p = new Women();
Women w = (Women)p;
w.buy();
```

可能发生的异常：ClassCastException — 类型转换异常

● 如何防止类型转换异常？

答：instanceof。

```
Person p = new Women();
if(p instanceof Women) {
    Women w = (Women)p;
    w.buy();
}
```

注意：属性没有多态性！ 因为属性不存在重写

```
public class Person {
    String name = "Person";
    void show() {
        System.out.println("Person.show()");
    }
}

public class Man extends Person {
    String name = "Man";
    @Override
    void show() {
        System.out.println("Man.show()");
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Man();
        System.out.println(p.name);
        p.show();

        if(p instanceof Man)
        {
            Man m = (Man)p;
            System.out.println(m.name);
            m.show();
        }
    }

    Person
    Man.show()
    Man
    Man.show()
}

结果：
```

看看上述代码在 C++ 中的运行情况：

```
#include <iostream>
```

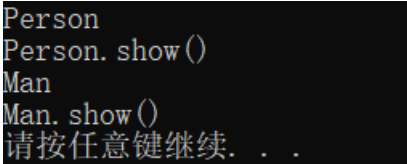
```

class Person
{
public:
    char* name = "Person";
    void show();
};
void Person::show()
{
    std::cout << "Person.show()" << std::endl;
}
class Man : public Person
{
public:
    char* name = "Man";
    void show();
};
void Man::show()
{
    std::cout << "Man.show()" << std::endl;
}

int main()
{
    Person *p = new Man();
    std::cout << p->name << std::endl;
    p->show();

    Man *m = (Man*)p;
    std::cout << m->name << std::endl;
    m->show();
}

```



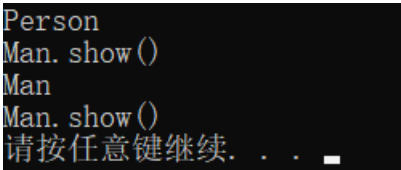
```

Person
Person.show()
Man
Man.show()
请按任意键继续. . .

```

结果:

把 Person 中的方法改成虚方法: `virtual void show();`



```

Person
Man.show()
Man
Man.show()
请按任意键继续. . .

```

结果:

11 第八天：P190-P193 面向对象的第四大特征：抽象性

abstract 修饰的方法：抽象方法

- 1、抽象方法没有方法体。
- 2、抽象方法所在的类必须是抽象类。

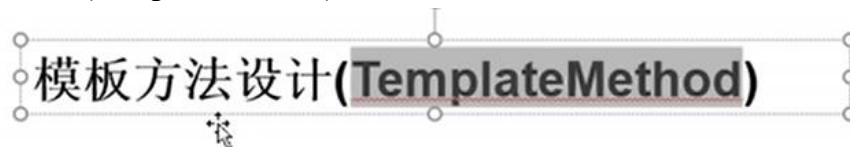
abstract 修饰的类：抽象类

- 1、抽象类不能够显示的调用其构造器来实例化。
- 2、必须由其子类来重写抽象类的全部抽象方法后，才可以实例化。
- 3、抽象类是有构造器的，并且要由其子类的构造器来隐式调用。
- 4、抽象类中可以有非抽象方法。也可以没有一个抽象方法。但是这样就没有什么意义。
- 5、抽象类的子类可以还是抽象类，只实现了部分方法或完全没有实现。

abstract 不能修饰属性、私有方法、静态方法、代码块、构造器、final 方法和 final 类。

abstract 不能和 final、private、static 一起使用。

11.1.1 模板方法(TemplateMethod)设计模式



抽象类体现的就是一种模板模式的设计，抽象类作为多个子类的通用模板，子类在抽象类的基础上进行扩展、改造，但子类总体上会保留抽象类的行为方式。

解决的问题：

- 当功能内部一部分实现是确定，一部分实现是不确定的。这时可以把不确定的部分暴露出去，让子类去实现。
- 编写一个抽象父类，父类提供了多个子类的通用方法，并把一个或多个方法留给其子类实现，就是一种模板模式。

```
abstract class CodeRunTimeTest {  
    /*  
     * 测试方法  
     */  
    public long countRunTime()  
    {  
        // 1.记录开始运行的时间  
        long start = System.currentTimeMillis();  
        // 2.执行被测代码  
        runCode();  
        // 3.记录运行结束的时间  
        long end = System.currentTimeMillis();  
    }  
}
```



```

        // 4.返回时间差
        return (end - start);
    }
    /*
     * 被测方法，由子类重写
     * C#中的委托
     */
    protected abstract void runCode();
}

public class Main {
    public static void main(String[] args) {
        // 创建了一个有名抽象类的匿名子类的有名对象
        CodeRunTimeTest test = new CodeRunTimeTest() {
            @Override
            protected void runCode() {
                System.out.println("重写了被测方法");
            }
        };
        System.out.println(test.countRunTime());
    }
}

```

11.1.2 匿名子类（匿名内部类）

如上：Main 类的 main 方法中创建了一个继承自 CodeRunTimeTest 类的子类，**该子类没有名字**，叫做匿名子类。而匿名的子类的对象有名字。对于接口也有匿名实现类。

使用匿名类时，必然是在某个类中直接用匿名类创建对象，因此匿名类一定是内部类，所有又叫做匿名内部类。

匿名对象：new CodeRunTimeTest ()，一般用于方法的实参或只调用一次对象的实例方法。**创建的对象没有名字**。而该对象的类有名字 CodeRunTimeTest。

匿名子类的匿名对象：<https://blog.csdn.net/u013771764/article/details/82937903>

```

jumpass.addKeyListener
(
    // 匿名内部类 - 键盘适配器
    // 创建了一个有名(KeyAdapter)抽象类的匿名子类的匿名对象
    // 作为 addKeyListener 方法的实参
    new KeyAdapter()
    {
        // 键入某个键时调用此方法
        public void keyTyped(KeyEvent e)
        {
            // 如果键入的字符是 0~9，或者按键是 Del 键或 Backspace 键，则
            // 直接返回读入的键盘字符，否则，设置键入的字符为键位未知 (0)
            if (((e.getKeyChar() <= 0x39) && (e.getKeyChar() >= 0x30)) ||
                (e.getKeyChar() == 127) || (e.getKeyChar() == 8))

```

```

        {
            e.setKeyChar(e.getKeyChar());
        }
        else
        {
            e.setKeyChar((char) 0);
        }
    }
}
);

```

12 第九天：P172-P176 Object 类

Object 类是所有 Java 类的根父类。

如果一个类没有显示地继承某个类，那么这个类会默认继承 Object 类。

Object 类有一个空参构造器。

Object 类中有 11 个方法，其中有三个重载方法 wait。

```

public class Object {
    private static native void registerNatives();
    static {
        registerNatives();
    }
    public final native Class<?> getClass();
    public native int hashCode();
    public boolean equals(Object obj);
    protected native Object clone();
    public String toString();
    public final native void notify();
    public final native void notifyAll();
    public final native void wait(long timeout);
    public final void wait(long timeout, int nanos);
    public final void wait();
    protected void finalize();
}

```

12.1 equals 方法

【面试题】== 与 equals 的区别。

答：==：如果比较的是基本数据类型，那么比较的是变量的值（存在自动类型提升）。如果比较的是引用数据类型，那么比较的也是变量的值（地址值），即比较两个引用是否指向同一个对象。equals：是一个方法，可以被重写。在 Object 类中等于==。像 String、Date 等类都重写了 equals 方法，用来比较内容。

Object 类的 equals 方法比较两个引用所指向的是不是同一个对象。

```

public boolean equals(Object obj) {

```

```
    return (this == obj);
}
```

String 类重写了该方法：比较字符串的内容是否相同。

```
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;
}
```

12.2 toString 方法

输出一个引用时，实际上是先默认调用了 toString 方法得到一个字符串，然后调用重载方法 public void System.out.println(String x)。

```
//com.atguigu.java2.Animal@659e0bfd
System.out.println(animal.toString());
//com.atguigu.java2.Animal@659e0bfd
System.out.println(animal); //默认调用就是toString方法
```

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

13 第九天：P194-P200 接口

格式：

```
[public] interface 接口名称 [extends 其他的接口名, 其他的接口名 ... ] {
    // 声明常量
```

```
// 抽象方法  
}
```

- 1、接口和类之间是并列关系。
- 2、接口不能被实例化，接口没有构造器。
- 3、接口中只能有常量和抽象方法（jdk1.8 之前）。
常量：`public static final double PI = 3.14;` (public static final 可以省略，这三者无论怎样组合，最终都会被补全。如：final, static, public, static final, public static, public final。抽象方法和静态方法同理。)
抽象方法：`public abstract void fly();` (public abstract 可以省略)
- 4、jdk1.8 新特性：接口中还可以有静态方法和默认方法。
静态方法：`public static void show(){} (public 可以省略，一定是 public 的)`
默认方法：`public default void show(){} (public 可以省略，一定是 public 的)`
静态方法调用：接口名.静态方法名；默认方法调用：实现类的对象名.默认方法名。
???这不就是抽象类了么。。。
- 5、接口和接口之间的关系：接口 extends 接口，可以多继承。
- 6、类和接口之间的关系：类 implement 接口，可以实现多个接口。
- 7、一个非抽象类实现接口后，必须重写所有接口的**所有**抽象方法。如果没有全部重写，则需把类声明为抽象类。
- 8、接口只能被 public、abstract 修饰：前者控制接口的访问范围：public 或者缺省的。abstract 可以省略。<https://blog.csdn.net/sanpangouba/article/details/87349241>
- 9、接口主要用来定义规范，解除耦合关系。
- 10、父类和接口中有相同的方法：类优先原则。
- 11、两个接口中有相同的抽象方法，那么实现类只需重写一次，即同时重写这两个方法。
- 12、如果两个接口有相同的默认方法，实现类必须重写该方法；若不重写，则编译报错。

在实现类的方法中调用接口的默认方法：接口名.super.默认方法名()。

```
public class Main {  
    public static void main(String[] args) {  
        new C().say();  
    }  
}  
  
interface A {  
    public default void say() {  
        System.out.println("A");  
    }  
}  
  
interface B {  
    public default void say() {  
        System.out.println("B");  
    }  
}  
  
class C implements A, B {  
    @Override
```

```
public void say() {
    A.super.say();
    B.super.say();
    System.out.println("C");
}
}

A
B
C
结果:
```

接口名 变量名 = new 接口实现类(); // 接口类型的引用可以指向接口实现类的对象。
同样，接口也可以和抽象类一样，有匿名实现类。

No.	区别点	抽象类	接口
1	定义	包含抽象方法的类	主要是抽象方法和全局常量的集合
2	组成	构造方法、抽象方法、普通方法、常量、变量	常量、抽象方法、(jdk8.0:默认方法、静态方法)
3	使用	子类继承抽象类(extends)	子类实现接口(implements)
4	关系	抽象类可以实现多个接口	接口不能继承抽象类，但允许继承多个接口
5	对象	都通过对象的多态性产生实例化对象	
6	局限	抽象类有单继承的局限	接口没有此局限
7	实际	作为一个模板	是作为一个标准或是表示一种能力
8	选择	如果抽象类和接口都可以使用的话，优先使用接口，因为避免单继承的局限	

Java 8中关于接口的改进

Java 8中，你可以为接口添加**静态方法**和**默认方法**。从技术角度来说，这是完全合法的，只是它看起来违反了接口作为一个抽象定义的理念。

静态方法：使用 **static** 关键字修饰。可以通过接口直接调用静态方法，并执行其方法体。我们经常在相互一起使用的类中使用静态方法。你可以在标准库中找到像 Collection/Collections或者Path/Paths这样成对的接口和类。

默认方法：默认方法使用 **default** 关键字修饰。可以通过实现类对象来调用。我们在已有的接口中提供新方法的同时，还保持了与旧版本代码的兼容性。
比如：java 8 API中对Collection、List、Comparator等接口提供了丰富的默认方法。

14 第十天：P203-P205 枚举类

枚举类：一个类的对象的数量是可数的有限个数，这样的类叫做枚举类。(单例模式?)

14.1 自定义枚举类

jdk1.5 以前我们这样玩：

```
public class Main {
    public static void main(String[] args) {
        Season season = Season.SPRING;
        System.out.println(season.value() + " " + season.name());
        season = Season.WINTER;
        System.out.println(season.value() + " " + season.name());
    }
}
/* 自定义枚举类 */
final class Season {
    // 3.无法修改的属性：名字和值
    private final String name;
    private final int value;
    // 1.私有化构造器
    private Season(String name, int value) {
        this.name = name;
        this.value = value;
    }
    // 2.创建本类的可数个对象
    public static final Season SPRING = new Season("春天", 0);
    public static final Season SUMMER = new Season("夏天", 1);
    public static final Season AUTUMN = new Season("秋天", 2);
    public static final Season WINTER = new Season("冬天", 3);
    // 4.属性的同名 get 方法
    public String name() {
        return name;
    }
    public int value() {
        return value;
    }
}

    0 春天
结果： 3 冬天
```

14.2 enum 关键字

jdk1.5 引入了 enum 关键字，我们就可以这样玩了：

```
enum Season{
    SPRING, SUMMER, AUTUNM, WINTER;
    private Season() {}
}
```

创建枚举类的对象必须放在枚举类的类体的首行，多个对象之间用逗号(,)隔开，以分号结尾。(如果类体中只有枚举类的对象，分号可以省略；但如果还有其他结构如(构造器)，则一定要分号。)

枚举类的构造器一定是私有的(private)，默认送一个空参的私有的构造器。

枚举类中可以加上私有的属性和有参构造器：

```
enum Season{
    SPRING("春天"), // 这样子调用有参构造器, C++: ???
    SUMMER("夏天"),
    AUTUNM("秋天"),
    WINTER("冬天");
    // 私有属性
    private final String seasonName;
    private Season(String name) {
        this.seasonName = name;
    }
    public String getSeasonName() {
        return seasonName;
    }
}
```

14.3 枚举类中的常用方法

enum 关键字定义的枚举类会帮我们自动继承 Enum 类，并创建我们指定名字的那些本类的静态对象。其底层原理和前面的自定义枚举类原理差不多。

可以看看这篇 <https://blog.csdn.net/javazejian/article/details/71333103>

其父类 Enum 中提供了一些方法。

values()方法：返回枚举类型的对象数组。该方法可以很方便地遍历所有的枚举值。

valueOf(String str)方法：可以把一个字符串转化为对应的枚举类对象。要求字符串必须是枚举类对象的名字。如果不是，则会抛出 IllegalArgumentException 异常

14.4 实现接口的枚举类

枚举类可以用来实现接口。

它可以像普通的类一样实现接口：

```
public class Main {
    public static void main(String[] args) {
        Season season = Season.SPRING;
        season.info();
    }
}

interface I {
```

```
    void info();
}
enum Season implements I{
    SPRING,
    SUMMER,
    AUTUNM,
    WINTER;
    @Override
    public void info() {
        System.out.println("Season - I.info()");
    }
}
```

枚举类的每个对象都可以单独实现接口的方法：对象所实现的方法里面可以使用 `super` 来调用枚举类所实现的方法。

```
interface I {
    void info();
}
enum Season implements I{
    SPRING{
        @Override
        public void info() {
            super.info();
            System.out.println("spring info()");
        }
    },
    SUMMER{
        @Override
        public void info() {
            System.out.println("summer info()");
        }
    },
    AUTUNM{
        @Override
        public void info() {
            System.out.println("autunm info()");
        }
    },
    WINTER{
        @Override
        public void info() {
            System.out.println("winter info()");
        }
    }
};
```



```

@Override
public void info() {
    System.out.println("Season - I.info()");
}
}

```

15 第十天：P206-P207 注解（Annotation）

- 从 JDK 5.0 开始, Java 增加了对元数据(MetaData)的支持, 也就是 Annotation(注解)
- Annotation 其实就是代码里的**特殊标记**, 这些标记可以在编译, 类加载, 运行时被读取, 并执行相应的处理。通过使用 Annotation, 程序员可以在**不改变原有逻辑**的情况下, 在源文件中嵌入一些补充信息。
- Annotation 可以像**修饰符**一样被使用, 可用于**修饰包, 类, 构造器, 方法, 成员变量, 参数, 局部变量的声明**, 这些信息被保存在 Annotation 的 “name=value” 对中。
- 在 JavaSE 中, 注解的使用目的比较简单, 例如标记过时的功能, 忽略警告等。在 JavaEE/Android 中注解占据了更重要的角色, 例如用来配置应用程序的任何切面, 代替 java EE 旧版中所遗留的繁冗代码和 XML 配置等。

15.1 jdk 内置的三个常用注解

- 使用 Annotation 时要在其前面增加 @ 符号, 并把该 Annotation 当成一个修饰符使用。用于修饰它支持的程序元素
- 三个基本的 Annotation:
 - @Override**: 限定重写父类方法, 该注解只能用于方法
 - @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

@Override: 限定重写父类或者接口的方法, 该注解只能用于方法。

@Deprecated: 用于表示某个程序结构(类、方法、属性、构造器等)已过时。

@SuppressWarnings: 抑制编译器警告, 需要参数, 参数是一个字符串数组。

@SuppressWarnings("unused") : The value of the local variable *a* is not used

15.2 自定义注解

格式: @interface 注解名 {
}

```

@interface MyAnn {
    // 可以认为是声明了一个属性
    String name();
    // 属性可以有默认值
    int age() default 20;
}
@MyAnn(name = "111", age = 18)
class SuperClass {
    @MyAnn(name = "111")
    public void show() {

    }
}

```

15.3 元注解

元注解：可以理解成注解上的注解。元注解是用来给注解进行补充说明的。

JDK5.0提供了专门在注解上的注解类型，分别是：

- **Retention:保留策略**
- **Target: 目标**，可以用来修饰什么数据
- **Documented: 能否在生成的帮助文档中显示**
- **Inherited: 注解是否具备继承性**

● **@Target**：目标，用来指定注解可以使用在说明类型的结构上：属性、方法、类等。

```

@Target(ElementType.METHOD) // 该注解只能用于方法上
@interface MyAnn3 {

}

```

ElementType 是一个枚举，其源码(jdk1.8)如下：

```

public enum ElementType {
    TYPE, /** 类,接口(包括注解类型),或枚举声明 */
    FIELD, /** 属性声明(包含枚举常量) */
    METHOD, /** 方法声明 */
    PARAMETER, /** 形参声明 */
    CONSTRUCTOR, /** 构造器声明 */
    LOCAL_VARIABLE, /** 局部变量声明 */
    ANNOTATION_TYPE, /** 注解类型声明 */
    PACKAGE, /** 包声明 */
    TYPE_PARAMETER, /** Type parameter declaration @since 1.8 */
    TYPE_USE /** Use of a type @since 1.8 */
}

```

- **@Retention**:保留策略。注解保留到什么时候可以用,源文件,字节码文件,运行时。一般自定义注解使用 **RUNTIME**。

```
@Retention(RetentionPolicy.SOURCE)
@interface MyAnn3 {

}
```

RetentionPolicy 是一个枚举, 其源码(jdk1.8)如下:

```
public enum RetentionPolicy {
    SOURCE, /** 注解将被编译器丢弃。*/
    CLASS, /** 注解由编译器记录在类文件中,但是在运行时不需要被 VM 保留。这是默认的*/
    RUNTIME /** 注解将被编译器记录在类文件中在运行时被 VM 保留,因此可以利用反射特性来读取它们。*/
}
```

16 第十天: P208-P218 异常体系结构

16.1 Exception 和 Error

异常: 在 Java 语言中, 将程序执行过程中发生的不正常情况称为异常。(开发过程中的语法错误和逻辑错误不是异常)

Error 错误: Java 虚拟机无法解决的严重问题。如, JVM 系统内部错误、资源耗尽等严重情况。比如, **StackOverflowError** 和 **OOM** (什么是 OOM? **OutOfMemoryError**: 程序申请内存过大, 虚拟机无法满足我们, 然后自杀了)。一般不编写针对性代码进行处理。

Exception 异常: 其他因编程错误或偶然的外在因素导致的一般性问题, 可以使用针对性的代码进行处理。

两种堆栈错误:

```
// StackOverflowError
main(args);
// OutOfMemoryError
int[] p = new int[1024 * 1024 * 1024];
```

异常体系的类关系:

```
|-- Throwable
    |-- Error
    |-- Exception
        |-- 编译时异常: 除了 RuntimeException (必须进行处理, 否则不能运行)
            IOException
                |-- FileNotFoundException
        |-- 运行时异常: RuntimeException
            |-- NullPointerException
            |-- IndexOutOfBoundsException
                |-- ArrayIndexOutOfBoundsException
                |-- StringIndexOutOfBoundsException
```

```
|-- IllegalArgumentException
|-- ClassCastException
|-- ...
```

16.2 try – catch – finally

```
try {
    可能会发生异常的代码
} catch(异常类型 1 e) {
    类型 1 异常处理代码
} catch(异常类型 2 e) {
    类型 2 异常处理代码
}
...
catch(Exception e) { // 一般这样写，以防止未考虑到的异常出现。(多态)
    类型 n 异常处理代码
} finally {
    离开 try-catch 结构时一定会执行的代码
}
```

在 try 语句块中的代码，一旦某一行代码发生异常，系统将会创建相应的异常对象，并抛出，**后面的 try 语句块中的代码就不会执行了。**

然后根据 catch 中的异常类型从上往下依次进行匹配，一旦匹配成，则执行异常处理代码。然后执行 finally 语句块，离开 try-catch 结构。**继续执行 try-catch 结构后面的代码。**

如果都匹配失败则程序终止运行，并且**终止之前会执行 finally 语句块。**

- 1、try 语句块中发生异常的那一行代码后面的代码不会执行了
- 2、无论是否发生异常，无论是否抓到异常，离开 try-catch 结构时 finally 语句块的代码一定被执行。就算 catch 中再次发生异常，finally 仍然会执行。即使 catch 中有 return，还是会执行。
- 3、如果正常的处理了异常，那么会继续执行 try-catch 结构后面的代码。
- 4、多个 catch 语句，子类在上，父类在下。
- 5、catch 和 finally 语句都是可选的。但至少要有其中一个，不能单单只有 try。
- 6、获取依次信息：getMessage()、printStackTrace()。

看看这个，牛逼咯！

```
public class FinallyTest {
    public static void main(String[] args) {
        int value = getRetrunValue();
        System.out.println(value);
    }

    public static int getRetrunValue() {
        try {
```

```
        System.out.println(1 / 0);
    } catch (Exception ex) {
        return 10;
    } finally {
        return 20;
    }
}
```

结果：
20

原因：函数的信息存储在栈帧中，函数的返回值也是放在栈帧的一个位置上。return 语句会将返回值放在该位置上，并终止本层函数，返回上一层调用。return 了两次，后一次值覆盖了前一次值。

16.3 throws

格式：方法名(形参列表) throws 异常类型 1, 异常类型 2, ... { 方法体 }

throws 并没有真正处理掉异常，而是将异常向上抛出，抛给方法的调用者，由其处理。

如果父类被重写方法没有抛出异常，那么子类重写方法也不能抛出异常。

子类重写方法所抛出的异常不大于父类被重写方法所抛出的异常。

抓的第二种方式：throws

格式：方法名(形参列表) throws 异常类型, 异常类型 2, ..., {}

说明：

1. throws 并没有真正的处理异常，而是将异常向上抛，抛给方法的调用者。谁调用该方法谁处理这个异常

try-catch 和 throws 的区别？

1. try-catch 是真正的将异常处理掉

2. throws 并没有真正的处理异常，只是将异常向上抛给方法的调用者，最后还是 try-catch 真正的将异常处理掉

什么时候使用 throws？

在 main 方法中如果需要连续调用多个方法并传入参数，那么如果发生异常，该异常只能向上抛。抛给方法的调用者进行处理。

什么时候不能使用 throws？

1. 如果父类被重写的方法没有抛出异常，那么子类重写的方法也不能抛出异常
2. 子类重写的方法抛出的异常不大于父类被重写方法抛出的异常

16.4 throw

格式：throw 异常类对象；

一般 throw 编译时异常的时候，我们都使用 throws 将其抛给上层方法处理。

【面试题】throw 和 throws 的区别？

throw：是用来制造异常的，是用来向外抛出异常的。

throws：是用来捕获异常的，捕获到异常后自己不处理，将异常抛出给方法的调用者。

16.5 自定义异常类

```
/*
 * 自定义异常类：
 *
 * 1. 继承Exception(编译时异常)或RuntimeException(运行时异常)
 * 2. 提供两个构造器(一个空参，一个有参)
 * 3. 提供一个serialVersionUID，可以显示的声明也可以不声明(系统会自动分配一个)。但是建议显示声明一个
 */
public class MyException extends Exception{

    /**
     * 可以显示的声明也可以不声明(系统会自动分配一个)。但是建议显示声明一个
     */
    private static final long serialVersionUID = -7736287090372348682L;

    public MyException(){

    }

    public MyException(String message){
        super(message); //调用父类的构造器
    }
}
```

16.6 总结

总结：异常处理5个关键字



17 第十一天：P218-P256 Java 常用类

17.1 单元测试

```
import org.junit.Test;

/*
 * 单元测试：
 * 1. 导包 -> 项目上右键 -> Build Path -> Add Libraries -> JUnit -> JUnit4
 * 2. 在类中写一个无返回无参的方法
 * 3. 在该方法上加一个注解@Test
 * 4. 运行：在测试方法上 -> 右键 -> RunAs -> JUnit4 TestRunner -> 红条表示运算失败，绿条表示成功
 *
 * 注意：
 * 1. 单元测试的方法所在的类必须是public
 * 2. 方法必须为无参无返回值
 */
public class UnitTestTest {

    @Test
    public void test(){
        System.out.println(1 / 0);
    }
}
```

类必须是 public 的，方法也必须是 public 的。

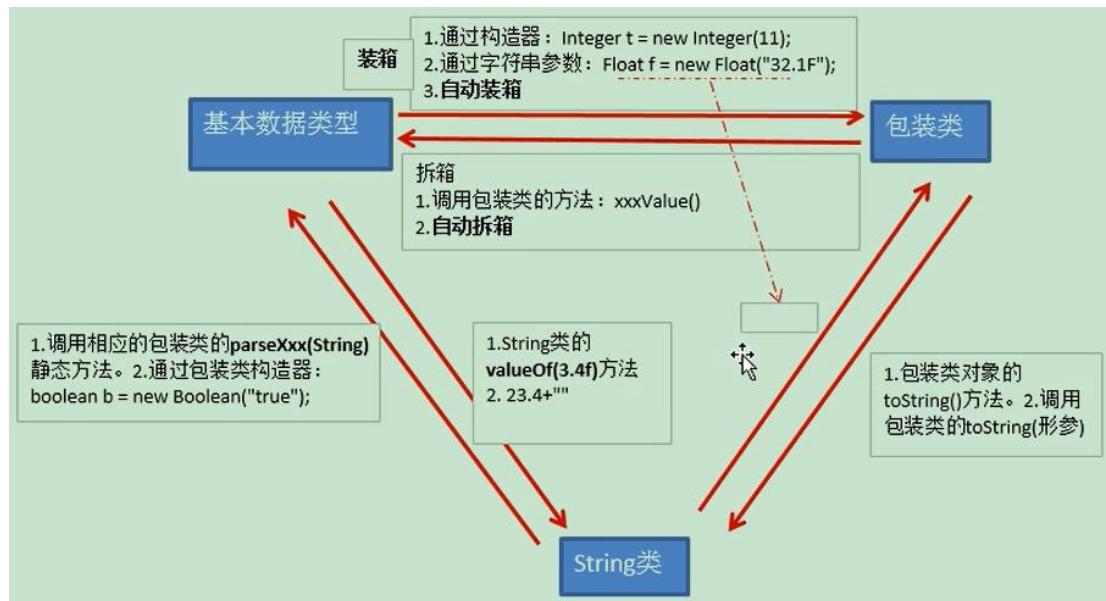
17.2 包装类(Wrapper)

基本数据类型	包装类
<u>boolean</u>	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

父类: Number

让天下没有难学的技术

17.3 基本数据类型、包装类、String 之间的转换



- 基本数据类型 → 包装类: 包装类的构造器, `Integer integer = new Integer(a);`
- 包装类 → 基本数据类型: 调用包装类的 `xxxValue()` 方法, `int a = integer.intValue();`
- 拆箱和装箱: 直接赋值, `Integer integer = a; int b = integer;`
- String → 基本数据类型: 调用包装类的 `parseXxx(String)` 静态方法, `int a = Integer.parseInt(str);`
- String → 包装类: 包装类的构造器, `Integer integer = new Integer(str);`
- 基本数据类型 → String: 调用 String 类的 `valueOf()` 静态方法, `String str = String.valueOf(10);`
- 包装类 → String: 调用包装类的 `toString()` 方法, `String str = String.valueOf(integer);`

拆箱: 将包装类直接赋值给基本数据类型。

装箱: 将基本数据类型直接赋值给包装类。底层是调用了 `valueOf` 方法。

```
/*
 * 拆箱: 将包装类直接赋值给基本数据类型
 * 装箱: 将基本数据类型赋值给包装类
 */
@Test
public void test4(){
    //拆箱
    int a = new Integer(10);

    //装箱
    Integer number = 30; //其实在底层是调用了valueOf方法

    //自动装箱了
    boolean equals = "aaa".equals(10);
}
```


注意 String 转 Boolean 时有个坑:只有 "true"(忽略大小写)转为 true, 其余(包括 null)转为 false。这里是源码:

```
public Boolean(String s) {
    this(parseBoolean(s));
}

public static boolean parseBoolean(String s) {
    return ((s != null) && s.equalsIgnoreCase("true"));
}
```

包装类可以自动拆箱, 然后自动类型提升, 然后自动装箱。看看这段代码的输出:

```
@Test
public void test01() {
    Object ob = true ? new Integer(1) : new Double(2.0);
    System.out.println(ob);
    System.out.println(ob.getClass());
}

1.0
class java.lang.Double
```

结果:

这个: 神奇的结果。

```
@Test
public void test02() {
    Integer i = new Integer(1);
    Integer j = new Integer(1);
    System.out.println(i == j);

    Integer m = 1;
    Integer n = 1;
    System.out.println(m == n);

    Integer x = 128;
    Integer y = 128;
    System.out.println(x == y);
}

false
true
false
```

结果:

原因: Integer m=1; 的底层是调用了 valueOf 方法。当 i 的值在 low(-128)到 high(127) 之间时, 是不会创建对象的。在此区间以外, 会创建新对象。

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
```

```

        return IntegerCache.cache[i + (-IntegerCache.Low)];
        return new Integer(i);
    }

```

类似的，Short 也是：

```

public static Short valueOf(short s) {
    final int offset = 128;
    int sAsInt = s;
    if (sAsInt >= -128 && sAsInt <= 127) { // must cache
        return ShortCache.cache[sAsInt + offset];
    }
    return new Short(s);
}

```

17.4 String 类及其相关类

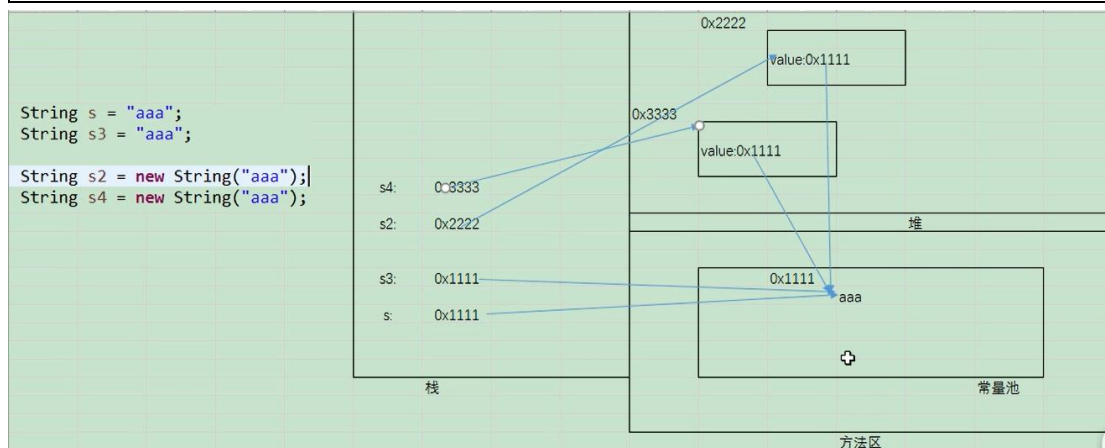
17.4.1 String 类和常量池

- 1、String 类被 final 所修饰，所有该类不能被继承。
- 2、实现了 Serializable 接口可以被序列化，被序列化后才能在不同的进程间或前后端进行数据传输。
- 3、实现了 Comparable 接口可以用来比较内容。
- 4、实现了 CharSequence 接口可以用来获取字符串长度，可以获取字符串中的某个字符。
- 5、String 对象的创建：String s = new String("aaa");。
- 6、字符串都放在常量池中：

```

@Test
public void test01() {
    String s1 = "aaa";
    String s2 = "aaa";
    String s3 = new String("aaa");
    String s4 = new String("aaa");
    System.out.println(s1 == s2);
    System.out.println(s3 == s4);
}

```



【面试题】String s = new String("aaa");在内存中创建了几个对象？

答：如果"aaa"已经在常量池中创建了，那么创建了 1 个对象；如果"aaa"没有在常量池

中创建过，那么创建了两个对象。堆中一个，常量池中一个。

7、String 的底层是一个数组：`private final char value[]`；该数组被 `final` 所修饰，所以 String 是一个不可变的字符序列。修改字符串时不会在原来的字符串对象上修改，而是会创建一个新的字符串。

这个有趣：

表 17-1

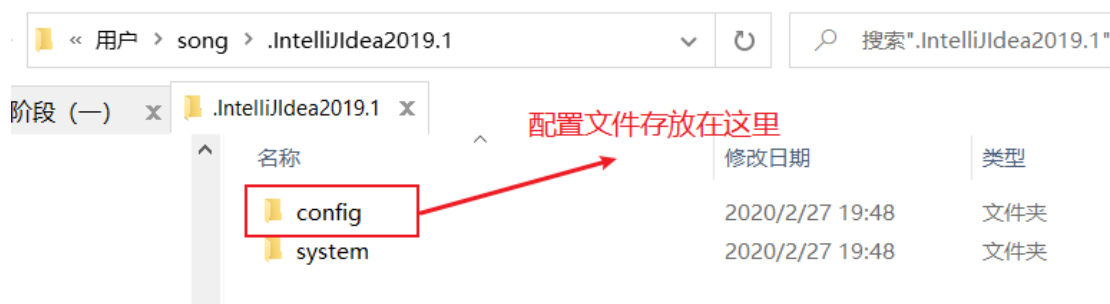
```
@Test
public void test02() {
    String s = "hellojava";
    String s1 = "hello";
    String s2 = "java";
    // 编译时就会将字符串拼接，和 s 没有区别
    String s3 = "hello" + "java";
    // 只有有变量参与字符串拼接，
    // 那么就会调用 StringBuilder 中的 toString 方法
    // 创建一个新的 Sting 对象。
    String s4 = s1 + "java";
    String s5 = "hello" + s2;
    String s6 = s1 + s2;
    // 直接去内存中的常量池中获取该字符串对象
    String s7 = s6.intern();
    System.out.println(s == s3); // true
    System.out.println(s == s4); // false
    System.out.println(s == s5); // false
    System.out.println(s == s6); // false
    System.out.println(s4 == s5); // false
    System.out.println(s4 == s6); // false
    System.out.println(s == s7); // true
}
```

17.4.2 String 类常用 API

- `public int length()`
- `public char charAt(int index)`
- `public boolean equals(Object anObject)`
- `public int compareTo(String anotherString)`
- `public int indexOf(String s)`
- `public int indexOf(String s, int startpoint)`
- `public int lastIndexOf(String s)`
- `public int lastIndexOf(String s, int startpoint)`
- `public boolean startsWith(String prefix)`
- `public boolean endsWith(String suffix)`
- `public boolean regionMatches(int firstStart, String other, int otherStart, int length)`

18 IDEA 的使用

用户更改设置的配置文件存在 config 文件夹下面



bin: 容器，执行文件和启动参数等。

help: 快捷键文档和其他帮助文档。

jre64: 64 位 java 运行环境。

lib: idea 依赖的类库。

license: 各个插件许可。

plugin: 插件。

1. 大家根据电脑系统的位数，选择 32 位的 VM 配置文件或者 64 位的 VM 配置文件。
2. 32 位操作系统内存不会超过 4G，所以没有多大空间可以调整，建议不用调整了。
3. 64 位操作系统中 8G 内存以下的机器或是静态页面开发者是无需修改的。
4. 64 位操作系统且内存大于 8G 的，如果你是开发大型项目、Java 项目或是 Android 项目，建议进行修改，常修改的就是下面 3 个参数：

`-Xms128m`，16 G 内存的机器可尝试设置为 `-Xms512m`
(设置初始的内存数，增加该值可以提高 Java 程序的启动速度。)

`-Xmx750m`，16 G 内存的机器可尝试设置为 `-Xmx1500m`
(设置最大内存数，提高该值，可以减少内存 Garbage 收集的频率，提高程序性能。)

`-XX:ReservedCodeCacheSize=240m`，16G 内存的机器可尝试设置为
`-XX:ReservedCodeCacheSize=500m`
(保留代码占用的内存容量。)