目录 Content

Java 入门

Java Applet 操作

1.编译 Java Applet

在命令行中使用 javac 命令，例如

C:\javawork>javac HelloWorldApplet.java

2. 运行 Java Applet

Java applet 不能直接执行，需要编写 HTML 文件，把 Applet 嵌入其中，如编辑 example.html 文件：

```
<HTML>
    <applet code="HelloWorldApplet.class"
            width=200 height=200>
     </applet>
</HTML>
```

用 appletviewer 来运行，在命令行中敲入如下命令：

C:\javawork>appletviewer example.html

在支持 Java 的浏览器上运行，例如在 IE 或 Netscape 中浏览 example.html 文件

package 语句

包(package)是用于管理生成的 java 字节码文件(.class)文件，其对应于文件系统中的目录，目的是把编译产生的.class 文件放入该目录，例如：shipping.objects 对应于目录"path\shipping\objects"，其中 path 需在编译时指定，且该目录须存在；如果"shipping\objects"不存在则创建该目录，例如：

C:\javawork>javac –d c:\work HelloWorldApp.java

则生成的 HelloWorldApp.class 文件将放在

"c:\work\shipping\objects"中

如果程序中没有 package 语句，或者有 package 语句但是在编译时不用-d 选项，则生成的.class 文件放入当前目录。

Java 编码规范

包名：包名是全小写的名词，中间可以由点分隔开

例如：java.awt.event;

类名：首字母大写，通常由多个单词合成一个类名，要求每个单词的首字母也要大写

例如：class HelloWorldApp;

接口名：命名规则与类名相同，例如 interface Collection;

方法名：往往由多个单词合成，第一个单词通常为动词，首字母小写，中间的每个单词的首字母都要大写，例如：balanceAccount， isButtonPressed;

变量名：全小写，一般为名词，例如：length;

常量名：基本数据类型的常量名为全大写，如果是由多个单词构成，可以用下划线隔开，例如：int YEAR, int WEEK_OF_MONTH；如果是对象类型的常量，则是大小写混合，由大写字母把单词隔开。

Java 语言基础

数据类型的划分

简单数据类型包括：

整数类型（Integer）：byte， short， int， long

浮点类型（Floating）：float，double

字符类型（Textual）：char

布尔类型（Logical）：boolean

复合数据类型包括：

class

interface

数组

常量和变量

变量定义：

typeSpecifier varName[=value[,varName[=value]…];

如：int count; char c='a';

String s;

-变量名对大小写敏感；

-对长度没有限制

-变量必须先初始化再使用

eg: int i;

System.out.println(i);

常量定义：

final typeSpecifier varName=value[,varName[=value]…];

如：final int NUM=100;

-常量只能被赋值一次，一旦赋值后不能再修改；

布尔类型——boolean

布尔型数据只有两个值 true 和 false，且它们不对应于任何整数值。

boolean b=true;

字符类型——char

字符常量：

字符常量是用单引号括起来的一个字符，如'a'，'A'

字符型变量：

类型为 char，它在机器中占 16 位，其范围为 0～65535。字符型变量的定义如：

char c='a'; /*指定变量 c 为 char 型，且赋初值为'a'*/

整型数据

整型常量：

◇ 十进制整数

如 123，－456，0

◇ 八进制整数

以 0 开头，如 0123 表示十进制数 83，－011 表示十进制数－9。

◇ 十六进制整数

以 0x 或 0X 开头，如 0x123 表示十进制数 291，－0X12 表示十进制数－18。

◇ 二进制整数

以 0b 开头，如 0b1001 表示十进制数 9。

数据类型	所占位数	数的范围
byte	8	$-2^7 \sim 2^7-1$
short	16	$-2^{15} \sim 2^{15}-1$
int	32	$-2^{31} \sim 2^{31}-1$
long	64	$-2^{63} \sim 2^{63}-1$

浮点型(实型)数据

浮点型数据的常量

◇ 十进制数形式

由数字和小数点组成，且必须有小数点，如 0.123，1.23，123.0

◇ 科学计数法形式

如：123e3 或 123E3，其中 e 或 E 之前必须有数字，且 e 或 E 后面的指数必须为整数。

◇ float 型的值,必须在数字后加 f 或 F,如 1.23f。

数据类型	所占位数	数的范围
float	32	$3.4e^{-38} \sim 3.4e^{+38}$
double	64	$1.7e^{-308} \sim 1.7e^{+308}$

自动类型转换规则

整型,实型,字符型数据可以混合运算。运算中，不同类型的数据先转化为同一类型，然后进行运算，转换从低级到高级；

低----------------------------------->高

byte,short,char-> int -> long -> float -> double

强制类型转换

高级数据要转换成低级数据，需用到强制类型转换，如：

int i;

byte b=(byte)i; /*把 int 型变量 i 强制转换为 byte 型*/

运算符和表达式

运算符

基本的运算符按功能划分，有下面几类：

1 算术运算符： +，-，*，/，%，++，--。

例如：

3+2; a-b; i++; --i; 注：i=5; →++i=6 TRUE i++=6 FALSE

i++ 先赋值再运算 ++I 先运算再赋值

2 关系运算符： >，<，>=，<=，==，!=。

例如：

count>3; I==0; n!=-1;

3 布尔逻辑运算符： !，&&，II 。

例如：

flag=true; !(flag); flag&&false;

4 位运算符： >>，<<，>>>，&，I，^，~。

例如：

a=10011101; b=00111001；则有如下结果：

a<<3 =11101000;

a>>3 =11110011 a>>>3=00010011;

a&b=00011001; aIb=10111101;

~ a=01100010; a^b=10100100;

5 赋值运算符 =，及其扩展赋值运算符如+=，-=，*=，/=等。

例如：

i=3;

i+=3; //等效于 i=i+3;

6 条件运算符 ？：

例如：result=(sum==0 ？ 1 ：num/sum);

7 其它:

包括分量运算符・，下标运算符 []，实例运算符 instanceof，内存分配运算符 new，强制类型转换运算符(类型)，方法调用运算符 () 等。例如:

System.out.println("hello world");

int array1[]=new int[4];

&&和||被称为短路逻辑运算符

优先次序	运算符
1	. [] ()
2	++ -- ! ~ instanceof
3	new (type)
4	* / %
5	+ -
6	>> >>> <<
7	> < >= <=
8	= = !=
9	&
10	^
11	|
12	&&
13	||
14	?:
15	= += -= *= /= %= ^=
16	&= |= <<= >>= >>>=

表达式

表达式是由操作数和运算符按一定的语法形式组成的符号序列。

一个常量或一个变量名字是最简单的表达式，其值即该常量或变量的值;

表达式的值还可以用作其他运算的操作数，形成更复杂的表达式。

表达式的类型由运算以及参与运算的操作数的类型决定，可以是简单类型，也可以是复合类型:

布尔型表达式: x&&yllz;

整型表达式: num1+num2;

数学函数

java.lang.Math

常量

Math.PI

Math.E

常用方法

Math.sqrt( );

Math.pow(x,a);

……

注：如果在源程序的头部加上 import static java.lang.Math.*;

则不用加 Math.就可以直接调用其方法和常量；

控制语句

Java 中的控制语句有以下几类：

◇ 分支语句：if-else，switch

◇ 循环语句：while，do-while，for

◇ 与程序转移有关的跳转语句：break，continue， return

◇ 例外处理语句：try-catch-finally，throw

块作用域

块：由一对花括号括起来的若干简单的语句；

一个块可以嵌套在另一个块中；

不能在嵌套的两个块中声明同名的变量；

```
public static void main(String[] args)
    {
        int n;
        …
       {
          int k;
          int n;
          …
        }
    }
```

分支语句

条件语句 if-else

```
if(boolean-expression)
      statement1;
    [else statement2;]
```

多分支语句 switch

```
switch(expression){
        case value1:statement1;
                    break;
```

```
        case value2:statement2;
                        break;
            ……
         case valueN:statementN;
                        break;
        [default: defaultStatement;]
    }
```

◇表达式 expression 的返回值类型必须是这几种类型之一：int,byte,char,short，枚举常量，字符串（JAVA SE 7）。

◇ case 子句中的值 valueN 必须是常量，而且所有 case 子句中的值应是不同的。

◇ default 子句是可选的。

◇break 语句用来在执行完一个 case 分支后，使程序跳出 switch 语句，即终止 switch 语句的执行（在一些特殊情况下，多个不同的 case 值要执行一组相同的操作，这时可以不用 break）。

循环语句

while 语句

```
[initialization]
while (termination){
    body;
    [iteration;]
}
```

do-while 语句

```
[initialization]
do {
    body;
    [iteration;]
} while (termination);
```

for 语句

```
for (initialization; termination; iteration){
    body;
}
for( i=0, j=10; i<j; i++, j--){
        ……
    }
```

跳转语句

◇ break 语句

◇ continue 语句

◇ 返回语句 return

例外处理语句

包括 try,catch,finally,throw 语句;

```
try {
        可能产生例外的代码段
   }catch(例外 1){处理例外 1}
     catch(例外 2){处理例外 2}
   finally{}
```

用 throw 来抛出一个例外对象:

```
throw new exception
```

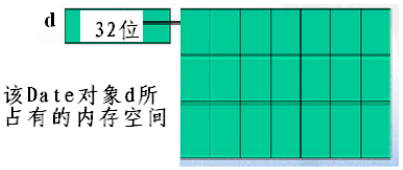
数组

引用(Reference)的概念

每个引用占据 32 位的内存空间，其值指向对象实际所在的内存中的位置，例如:

```
Date d=new Date();
```

通常我们称 d 是 Date 型的对象，实际上 d 就是引用，它是一个 32 位的数据，它的值指向 Date 对象实际所在的内存空间。



一维数组

一维数组的定义

type arrayName[ ];

类型(type)可以为 Java 中任意的数据类型，包括简单类型和复合类型。

例如:

```
int intArray[];
Date dateArray[];
        int []intArray;
Date []dateArray;
```

一维数组的初始化

◇ 静态初始化

```
int intArray[]={1,2,3,4};
String stringArray[]={"abc", "How", "you"};
```

◇ 动态初始化

1）简单类型的数组

```
int intArray[];
intArray = new int[5];→进行空间分配
```

2）复合类型的数组

```
Date dateArray[ ];
```

```
dateArray = new Date[3];
    /*为数组中每个元素开辟引用空间(32 位) */
dateArray[0]= new Date( );
    //为第一个数组元素开辟空间
dateArray[1]= new Date( );
    //为第二个数组元素开辟空间
dateArray[2]= new Date();
    // 为第三个数组元素开辟空间
```

一维数组元素的引用

数组元素的引用方式为：

arrayName[index]

index 为数组下标，它可以为整型常数或表达式，下标从 0 开始。每个数组都有一个属性 length 指明它的长度，例如：intArray.length 指明数组 intArray 的长度。

数组拷贝

引用拷贝：允许将一个数组变量拷贝给另一个数组变量，两个变量将引用同一个数组;

```
int[] smallPrimes = {2,3,5,7,11,13}
int[] luckyNumbers = smallPrimes;
```

值拷贝：一个数组的值拷贝到一个新的数组，用类的方法;

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length)
```

数组排序

Arrays 类中的 sort 方法：

```
int[] a = new int[];
Arrays.sort(a);
```

for each 循环

依次处理数组中的每个元素，而不必指定下标值

```
for(variable:collection) statement
```

定义一个变量用于暂存集合中的每一个元素，并执行相应的语句/语句块。

variable:暂存集合中元素的变量;

collection:数组或实现了接口的变量;

```
Eg:  for(int element : a)
        System.out.println(element);
```

打印数组 a 的元素，一个元素占一行;

多维数组

Java 语言中，多维数组被看作数组的数组。

二维数组的定义

```
type arrayName[ ][ ];
type [ ][ ]arrayName;
```

二维数组的初始化

◇ 静态初始化

```
int intArray[ ][ ]={{1,2},{2,3},{3,4,5}};
```

Java 语言中，由于把二维数组看作是数组的数组，数组空间不是连续分配的，所以不要求二维数组每一维的大小相同。

◇ 动态初始化

1) 直接为每一维分配空间，格式如下：

```
arrayName = new
type[arrayLength1][arrayLength2];
int a[ ][ ] = new int[2][3];
```

2) 从最高维开始，分别为每一维分配空间：

```
arrayName = new type[arrayLength1][ ];
arrayName[0] = new type[arrayLength20];
arrayName[1] = new type[arrayLength21];
…
arrayName[arrayLength1-1] = new type[arrayLength2n];
```

3) 例：

二维简单数据类型数组的动态初始化如下，

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

对二维复合数据类型的数组，必须首先为最高维分配引用空间，然后再顺次为低维分配空间。

而且，必须为每个数组元素单独分配空间。

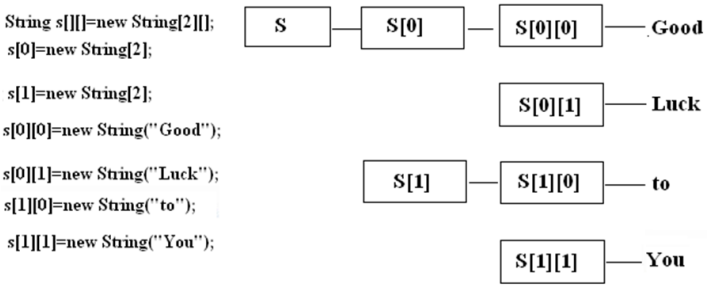
例如：

```
String s[ ][ ] = new String[2][ ];
s[0]= new String[2];
s[1]= new String[2];
s[0][0]= new String("Good");
s[0][1]= new String("Luck");
```

s[1][0]= new String("to");
s[1][1]= new String("You");

二维数组内存分配过程

引用方式为:arrayName[index1][index2];
如:num[1][0];



字符串

字符串的表示

Java 语言中，把字符串作为对象来处理，类 String 和 StringBuffer 都可以用来表示一个字符串

字符串常量

字符串常量是用双引号括住的一串字符。

"Hello World!"

String 表示不可变字符串

用 String 表示字符串：
String(char chars[ ]);
String(char chars[ ], int startIndex, int numChars);

String 使用示例：
String s=new String() ；生成一个空串
String greeting = “Hello”;
下面用不同方法生成字符串"abc"：
char chars1[]={'a','b','c'};
char chars2[]={'a','b','c','d','e'};
String s1=new String(chars1);
String s2=new String(chars2,0,3)

用 StringBuffer 表示字符串

StringBuffer( ); /*分配 16 个字符的缓冲区*/
StringBuffer(int len); /*分配 len 个字符的缓冲区*/
StringBuffer(String s); /*除了按照 s 的大小分配空间外,再分配 16 个字符的缓冲区*/

访问字符串

类 String 中提供了 length( )、charAt( )、indexOf( )、lastIndexOf( )、getChars( )、getBytes( )、toCharArray( )等方法。

类 StringBuffer 提供了 length( )、charAt( )、getChars( )、capacity()等方法。

修改字符串

String 是不可变字符串

指的是变量一旦被赋值，其值不能被改变，如果想改变变量的值，只能将变量引向另一个字符串

String greeting = “Hello”;

greeting = greeting.substring(0,3) + “p!”

String 类提供的方法：

concat( )

replace( )

substring( )

toLowerCase( )

toUpperCase( )

字符串比较

equals( )和 equalsIgnoreCase( )

它们与运算符'= ='实现的比较是不同的。运算符'= ='比较两个对象是否引用同一个实例，而 equals( )和 equalsIgnoreCase( )则比较两个字符串中对应的每个字符值是否相同。

空串与 Null 串

空串是长度为 0 的字符串

if(str.length()==0)或 if(str.equals(“”))

Null 串表示目前没有任何对象与该变量关联，即变量没有初始化

if(str == null)

注：原则上，对任何字符串变量进行操作前，都应该检查其是否是 Null 串，因为对 Null 串进行任何操作都是错误的。

字符串的转化

java.lang.Object 中提供了方法 toString( )把对象转化为字符串。

字符串“+”操作

运算符‘+’可用来实现字符串的连接：

String s = “He is ”+age+“ years old.”;

其他类型的数据与字符串进行“+”运算时，将自动转换成字符串。

具体过程如下：

String s=new String(new StringBuffer("he is").append(age).append("years old"));

简单输入输出

读取输入

-通过类 Scanner 从控制台进行输入

Scanner in = new Scanner(System.in);

输入一行

String name = in.nextLine();

读取一个单词(以空白符作为分隔符)

String firstName=in.next();

读取一个整数

int age = in.nextint();

读取一个浮点数

double e= in.nextDouble();

格式化输出

-System.out.print()

以 x 对应的数据类型所允许的最大非 0 数字位数打印输出 x,

-System.out.println()

-System.out.printf(String format,Object args...)

d 十进制整数

x 十六进制整数

o 八进制整数

s 字符串

c 字符

读取输入

-java.util.Scanner

Scanner(InputStream in)

String nextLine()

String next()

String nextInt()

String nextDouble()

boolean hasNext()

boolean hasNextDouble()

boolean hasNextInt()

Java 面向对象特性

面向对象思想

类的基本概念

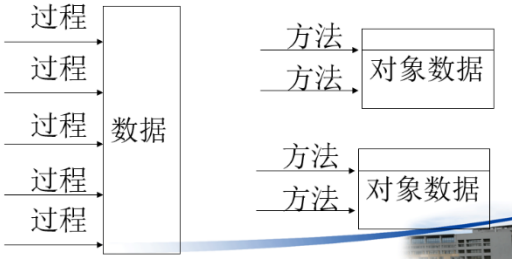
类是具有相同属性和服务的一组对象的集合，它为属于该类的所有对象提供了统一的抽象描述，其内部包括属性和服务两个主要部分。

算法+数据 or 数据+算法

对于规模较小的问题，使用过程化程序设计比较合适，但对于大规模问题，使用面向对象有以下好处：

-类提供了一种便于将众多的方法聚集在一起的机制；如 2000 个过程，可能需要 100 个类，平均每个类 20 个方法；

-类的封装机制有助于对其他的类方法隐藏数据操作；



面向对象

-先从项目中分离出类

-确定类中的成员及方法

类

类声明：

[public][abstract|final] class className [extends superclassName]
[implements interfaceNameList]
{……}

类体定义如下：

class className
{ [public | protected | private] [static]
[final] [transient] [volatile] type
variableName; //成员变量
[public | protected | private] [static]
[final | abstract] [native] [synchronized]
returnType methodName([paramList])
[throws exceptionList]
{statements} //成员方法
}

成员变量

[public | protected | private] [static]
[final] [transient] [volatile] type
variableName;

static: 静态变量（类变量）；相对于实例变量
final: 常量
transient: 暂时性变量，用于对象存档
volatile: 贡献变量，用于并发线程的共享

成员方法

方法的实现包括两部分内容：方法声明和方法体。
[public | protected | private] [static]
[final | abstract] [native] [synchronized]
returnType methodName([paramList])
[throws exceptionList] //方法声明
{statements} //方法体

方法声明中的限定词的含义：

static: 类方法，可通过类名直接调用
abstract: 抽象方法，没有方法体
final: 方法不能被重写
native: 集成其它语言的代码
synchronized: 控制多个并发线程的访问

对象

对象的三个主要特性：

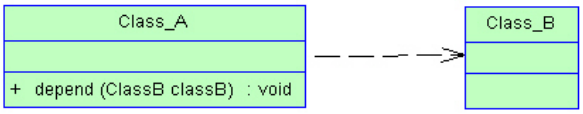
对象的行为(behavior)
可以对对象施加哪些操作，或可以对对象施加哪些方法?
对象的状态(state)
当施加那些方法时，对象如何响应?
对象的标识(identity)
如何区分具有相同行为与状态的不同对象?

同一个类的所有对象由于支持相同的行为而具有家族的相似性；
对象状态的改变必须通过对象的行为，即通过调用方法实现；
对象当前的状态限定了可以调用的方法；

类之间，最常见的关系：

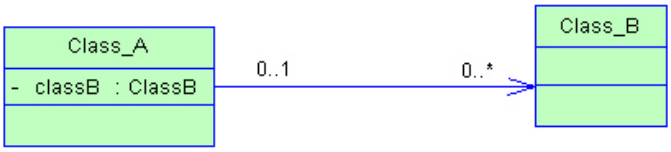
依赖

如果一个类的方法操纵另一个类的对象，就说一个类依赖于另一个类。这种使用关系是具有偶然性的、临时性的、非常弱的。表现在代码层面，为类 B 作为参数被类 A 在某个方法中使用。



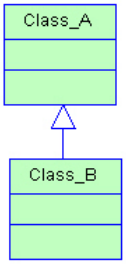
关联

体现的是两个类之间语义级别的一种强依赖关系；这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的、关联可以是单向、双向的；表现在代码层面，为被关联类 B 以类属性的形式出现在关联类 A 中。



继承

指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系；



特殊类的对象拥有其一般类的全部属性与服务，称作特殊类对一般类的继承。

例如，轮船、客轮；人、学生。

多继承：一个类可以是多个一般类的特殊类，它从多个一般类中继承了属性与服务。

例如，客轮是轮船和客运工具的特殊类。

在 java 语言中，通常我们称一般类为父类（superclass,超类），特殊类为子类(subclass)。

使用现有类

对象与对象变量

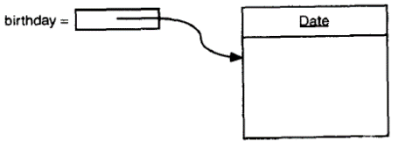
Java 使用构造器构造实例，构造器是一种特殊的方法，用来构造并初始化对象；

构造器的名字与类名相同；

```
new Date() ;
System.out.println(new Date());
String s = new Date().toString();
```

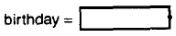
1. Date birthday = new Date( );

构造的对象被存放在一个变量 birthday 中；



2. Date deadline;

定义了一个对象变量，但目前这个对象变量没有指向任何对象。



s = deadline.toString( ); //wrong

1 一个对象变量并没有实际包含一个对象，而仅仅引用一个对象；
2 一个对象变量一定要引用一个用 new 创建的对象后，才能调用对象的方法；
3 可以显式地将对象变量设置为空，表明这个对象变量目前没有引用任何对象。等同于只是声明了一个对象变量，该对象变量没有指向任何对象。

```
deadline = null;
if(deadline != null)
    System.out.println(deadline);
```

应用 Java 库中的 LocalDate 类

JAVA 中表示日历的标准类；

构造对象（静态工厂方法 FACTORY METHOD）

```
LocalDate.now( )
LocalDate.of(1999,12,31)
LocalDate newYearsEve =LocalDate. of(1999,12,31);
```

更改器方法与访问器方法

```
LocalDate newYearsEve = LocalDate.of(1999,12,31);
int year = newYearsEve.getYear( );
Int month = newYearsEve.getMonthValue();
int day = = newYearsEve.getDayOfMonth();

LocalDate aThousandDaysLater = newYearsEve.plus(1000);
```

构造器方法

```
new GregorianCalendar( );
new GregorianCalendar(1999,11,31);
new GregorianCalendar(1999,Calendar.DECEMBER,31);
new GregorianCalendar(1999,12,31,23,59,59);
```

更改器方法与访问器方法

```
GregorianCalendar now = new GregorianCalendar( );
int month = now.get(Calendar.MONTH);
int weekday = now.get(Calendar.DAY_OF_WEEK);

deadline.set(Calendar.YEAR,2001);
deadline.set(Calendar.MONTH, Calendar.APRIL);
Deadline.set(Calendar.DAY_OF_MONTH, 15);
```

LocalDate 使用实例

生成一个日历

```
SUN MON TUE WED THR FRI SAT
                        1   2
  3   4   5   6   7   8   9
 10  11  12  13  14  15  16
 17  18  19  20  21  22  23
 24  25  26  27  28* 29  30
 31
```

实现自己的类

设计这个类的公共接口（服务）；

类:BankAccount
-存钱(Deposit money)
-取钱(Withdraw money)
-获取帐户余额(Get the current balance)

假设 harrysCheck 是这个类的一个对象,并且已经实例化:

```
    harryChecking.deposit(2000);
    harryChecking.withdraw(500);
    System.out.println(harrysChecking.
getBalance());
```

```
public void deposit(double amount)
public void withdraw(double amount)
public double getBalance()
```

设计这个类的构造函数;

构造函数是用来初始化对象的函数.
类似于成员方法,但有以下的不同:
-构造函数的名字必须与类的名字一样;
-构造函数没有返回值;

```
  public BankAccount(){
  //函数体
 }
```

```
public BankAccount()
  public BankAccount(double iniBalance)
```

```
public class BankAccount{
    public  BankAccount(){};
 public BankAccount(double iniBalance){};

 public void deposit(double amount){};
    public void withDraw(double amount){};
    public double getBalance(){};
}
```

简单应用

```
//Transfer from one account to another
Double transferAmount=500;
jackSaving.withDraw(transferAmount);
harryChecking.deposit(transferAmount);
//adds interest to a saving account
double interestRate=5;
Double interestAmount=harryChecking.getBalance()
*interestRate/100;
harryChecking.deposit(interestAmount);
```

对公共接口进行注释(文档注释)

```
Withaws money from a bank account.
        @param amount the amount to withdraw
    */
    public void withdraw(double amount){
    }

    /**
        Gets the current balance of the bank account.
        @return the current balance
    */
   public double getBalance(){
   }
```

添加成员变量

成员变量用来保存对象的状态.

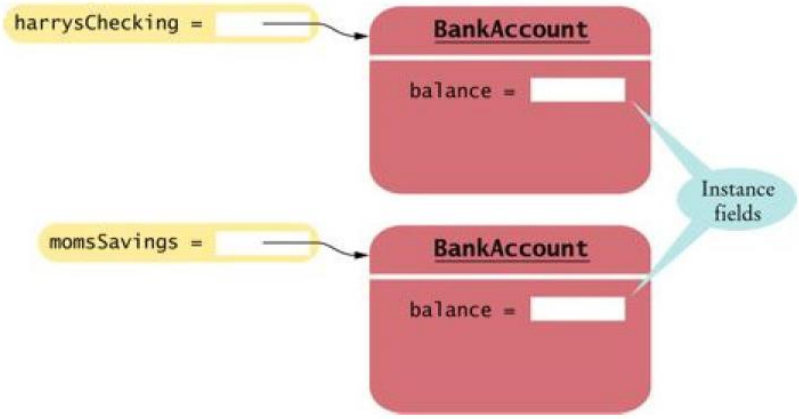
```
public class BankAccount{
 ……
 private double balance;
}
```

完成构造函数和公共接口函数

```
    public BankAccount(){
        balance=0;
  }

    public BankAccount(double iniBalance){
    balance=iniBalance;
}
```

BankAccount harryChecking=new BankAccount(1000);
-创建一个新的 BankAccount 对象变量;
-调用第二个构造函数;
-将值参 iniBalance 设为 1000
-将新创建对象的成员变量 balance 的值设为 iniBalance 的值;
-返回新创建对象的引用;
-将新创建对象的引用值赋给变量 harryChecking




```
public void deposit(double amount){
      double newBalance=balance+amount;
  balance=newBalance;
}
  public void withdraw(double amount){
      double newBalance=balance-amount;
  balance=newBalance
}
 public double getBalance(){
      return balance;
}
```

类的特性

隐式参数和显式参数

```
public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }
```

Employee number007= new Employee(“James Bond", 75000, 1963, 12, 15);
number007.raiseSalary(5); 隐式参数 显式参数

```
 double raise = number007.salary * byPercent / 100;
    number007.salary += raise;
```

```
public void raiseSalary(double byPercent)
  {
    double raise = this.salary * byPercent / 100;
    this.salary += raise;
  }
```

封装的优点

```
public String getName(){ return name;}
public double getSalary(){return salary;}
public Date getHireDay() { retrurn hireDay;}
```

类中如果需要获得或设置实例域的值:

一个私有的数据域;
一个公有的域访问器方法;
一个公有的域更改器方法;

final 实例域: 可以将实例域定义为 final

```
  class Employee{
     private final String name;
     .......}
```

构建对象时必须初始化这样的域，即确保在构造器执行完之后，这个域的值被设置;
在以后的操作中，不能对其进行修改;

final 一般用于修饰简单数据类型，或不可变类。

实例成员和类成员

用 static 关键字可以声明类变量和类方法，其格式如下:

```
    static type classVar;
    static returnType
          classMethod({paramlist}) {
    …
    }
```

如果在声明时不用 static 关键字修饰，则声明为实例变量和实例方法。

每个对象的实例变量都分配内存
通过该对象来访问这些实例变量，不同的实例变量是不同的。
类变量仅在生成第一个对象时分配内存，所有实例对象共享同一个类变量
类变量可通过类名直接访问，无需先生成一个实例对象
也可以通过实例对象访问类变量。

类成员

```
public class BankAccount{
    ……
    private double balance;
    private int accountNumber;
}
```

要求：帐户号从 1000 开始，顺序取值
？如何知道当前分配的最后一个帐户号是多少

```
public class BankAccount{
    ……
    private double balance;
    private int accountNumber;
    private static int lastAssignedNumber=1000;
}
public class BankAccount{
    public BankAccount()
    {
        accountNumber = ++lastAssingedNumber;
    }
    ……

}
```

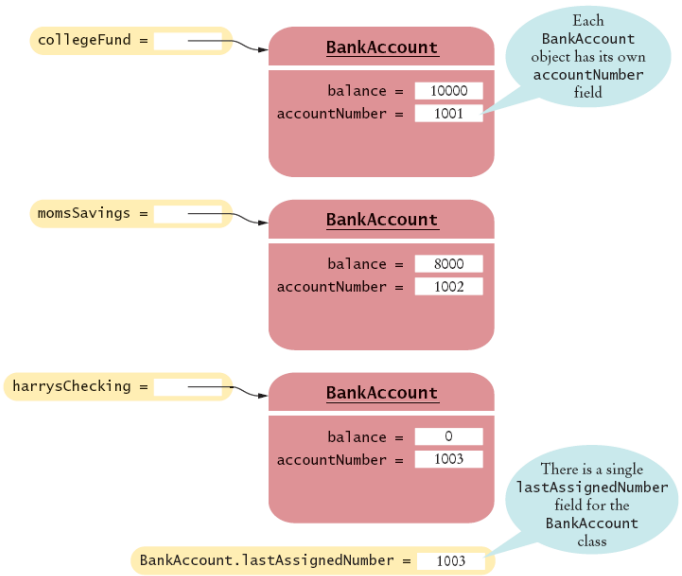


Figure 5 A Static Field and Instance Fields

实例方法和类方法

实例方法可以对当前对象的实例变量进行操作，也可以对类变量进行操作。

类方法不能访问实例变量，只能访问类变量。

类方法可以由类名直接调用，也可由实例对象进行调用。

类方法中不能使用 this 或 super 关键字。

```
class Member{
    3 个用法
    static int classVar;
    2 个用法
    int instanceVar;
    2 个用法
    static void setClassVar(int i){
        classVar = i;
        //instanceVar = i是错误的，类方法不能访问实例变量
    }
    2 个用法
    static int getClassVar(){
        return classVar;
    }
    2 个用法
    void setInstanceVar(int i){
        classVar = i;
        instanceVar = i;
    }
    2 个用法
    int getInstanceVar(){
        return instanceVar;
    }
        }
public class MemberTest {
    public static void main(String args[]){
        Member m1 = new Member();
        Member m2 = new Member();
        m1.setClassVar(1);
        m2.setClassVar(2);
        System.out.println("m1.ClassVar="+m1.getClassVar()+" m2.ClassVar="+m2.getClassVar());
        m1.setInstanceVar(11);
        m2.setInstanceVar(22);
        System.out.println("m1.InstanceVar=" +m1.getInstanceVar()+ " m2.InstanceVar="+m2.getInstanceVar());
    }
}
```

```
C:\Java\jdk-17.0.1\bin\java.exe "-javaa
m1.ClassVar=2 m2.ClassVar=2
m1.InstanceVar=11 m2.InstanceVar=22
```

方法参数

JAVA 当中参数传递采用的的是值传递的方法。

public static void tripleValue(double x)
{ x = 3 * x; }

double percent = 10;
tripleValue(percent);
precent 值没有发生变化

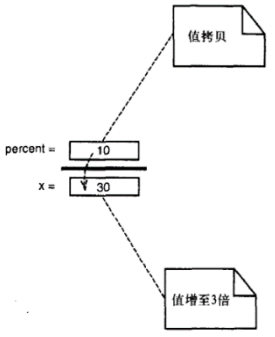
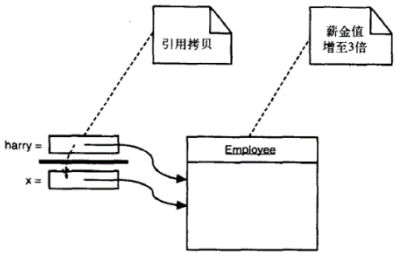


图4-6 对值参数的修改没有被保留下来

public static void tripleValue(Employee x)
{ x.raiseSalary(200); }
harry = new Employee(...);
tripleValue(harry);
harry 值发生了变化



public static void swap(Employee x, Employee y)//经典的交换例子
{ Employee temp = x;
x = y;
y = temp; }
Employee a = new Employee("Alice",...)
Employee b = new Employee("Bob",...);
swap(a,b);
JAVA 中方法参数采用值传递;
一个方法不能修改一个基本数据类型的参数;
一个方法可以改变一个可变类对象参数的状态;
一个方法 不能让对象参数引用一个新的对象。

对象构造

方法重载

方法重载是指多个方法享有相同的名字
区别在于:或者是参数的个数不同，或者是参数类型不同。
返回类型不能用来区分重载的方法。　　重写要求方法名一样和参数也要一样

```java
import java.io.*;

2 个用法
class MethodOverloading{
    1 个用法
    void receive(int i){
        System.out.print("Receive one int data ");
        System.out.println("i="+i);
    }
    1 个用法
    void receive(int x, int y){
        System.out.print("Receive two int datas ");
        System.out.println("x="+x+" y="+y);
    }
}
public class MethodOverloadingTest {
    public static void main(String args[]){
        MethodOverloading mo = new MethodOverloading();
        mo.receive( i: 1);
        mo.receive( x: 2, y: 3);
    }
}
```

```
C:\Java\jdk-17.0.1\bin\java.exe "-
Receive one int data i=1
Receive two int datas x=2 y=3
```

构造方法

◇ 重载经常用于构造方法。

◇ 构造方法具有和类名相同的名称，而且不返回任何数据类型。

◇Java 中的每个类都有构造方法，用来初始化该类的一个对象。

◇ 构造方法只能由 new 运算符调用

```
class Point{
    private int x,y;
    public Point(){
    x=0; y=0;
    }
    public Point(int x, int y){
    this.x=x;
    this.y=y;
        }
}
```

默认构造器/无参数构造器

◇默认构造器指没有参数的构造器。

```
public Employee()
    {
    }
```

◇ 如果在编写一个类时没有编写构造器，系统会提供一个默认构造器。该构造器将所有的成员变量/实例域设置为默认值。(数值型数据设置为 0，布尔型数据设置为 false，所有对象变量将设置为 null)

◇ 如果编写时提供了至少一个构造器，则系统不会提供默认构造器。

实例域初始化

◇在声明实例域时直接赋值;

◇在构造方法中赋值;

◇在初始化块中赋值;

◇如果实例域没有显式地赋予初值，则会被系统自动地赋为默认值：数值型数据设置为 0，布尔型数据设置为 false，所有对象变量将设置为 null。

◇在声明实例域时直接赋值/显示域初始化;

在执行构造器前，先执行该赋值操作，故当所有的构造器希望把相同的值赋给特定的实例域，可采用此方式。

```
class Employee { private String name = "" }
class Employee {
    private static int nextId;
    private int id = assignId();
    .....
}
```

◇初始化块：一个类的声明中可以包含多个代码块，只要构造类的对象，这些块就会被执行。

```
class Employee{
  private static int nextId;
  private int id;
   {
    id = nextId;
     nextId++;
    }
   ……}
```

◇调用另一个构造器

```
    this(...)
public Employee(double s) {
    //calls Employee(String, double)
    this("Employee #" + nextId, s);
     nextId++;
}
```

注：调用另一个构造器的语句必须是构造器中的第一条语句；

构造器初始化成员变量的步骤：

◇所有成员变量被初始化为默认值；

◇按照在类声明中出现的次序依次执行所有域初始化语句和初始化块；

◇如果构造器第一行调用了第二个构造器，则执行第二个构造器主体；

◇执行这个构造器的主体。

静态域初始化

◇通过提供一个初始化值

```
private static int nextId = 1;
```

◇静态的初始化块

```
static
 {
   Random generator = new Random();
   nextId = generator.nextInt(10000);
 }
```

类第一次加载的时候，会进行静态域的初始化。默认的初始值是 0,false,null。静态初始化语句以及静态初始化块按照类定义的顺序执行。

类设计技巧

成员变量一般设计为私有

数据的表示形式可能会改变,但其使用方式却不会经常发生变化

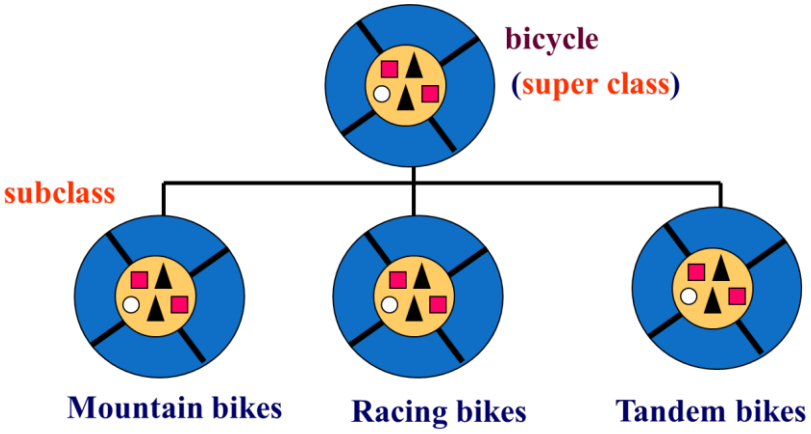
-一个私有成员变量
-一个公有的数据访问方法
-一个公有的数据更改方法

一定要对成员变量进行初始化
不要在类中使用过多的基本数据类型
将职责过多的类进行分解
类名和方法名要能够体现其职责

继承

继承

基于已存在的类（父类/超类/基类）构造一个新类（子类/派生类）。子类（继承）父类的域和方法，子类与父类是特殊与一般（is-a）的关系。



继承的定义形式

```
class Sub extends Base{
      ……
  }
```

Sub 类会自动拥有在 Base 类中定义的域和方法。

假设存在一个Employee类

```
public class Employee {
    public Employee(String n, double s, int year, int
                                                month, int day)   {
……}

    public String getName()   { …… }

    public double getSalary()   { …… }

    public Date getHireDay()   { …… }

    public void raiseSalary(double byPercent)   { …… }

    private String name;
    private double salary;
    private Date hireDay;
}
```

继承的实例（1）

- 现在需要定义Manager工种，因为Manager和Employee之间存在的“is - a”关系。所以可以将Manager设计为Employee的子类。

```
class Manager extends Employee {
    //添加方法和域
}
```

类的继承实例（2）

注意：子类拥有父类的所有域和方法。

增加域 / 方法

子类可以比父类拥有更多的域和方法。

- 例如：在Manager类中增加一个用于存储奖金信息的域，以及一个用于设置这个域的方法：

```
class Manager extends Employee {
    . . .
    public void setBonus (double b)
    {     bonus = b;     }

    private double bonus;
}
```

- Manager对象可以使用setBonus方法：

```
- Manager boss = ...;
- boss. setBonus(8888);
```

- setBonus方法没有在Employee中定义，所以Employee对象不能使用它。

- Manager 类自动继承了超类中的方法和域。

1. Manager 类继承的方法包括： getName、 getSalary、 getHireDay、raiseSalary。

2. Manager 类继承的域包括：name、salary、hireDay。但在 Manager 类中不能直接访问这些域。

在定义子类的时候，仅需要指出子类与超类的不同之处。因此类的设计原则之一是：

-尽量将通用的方法放置于超类之中；

-将具有特殊用途的方法放置在子类中。

方法的覆盖(override)

有些方法在子类 Manager 中并不适用。例如，Manager 类对象的总薪水是由薪水+奖金，所以 Manager 类中的 getSalary 方法应当返回薪水和奖金的总和(bonus + salary).

所以需要通过在 Manager 类中重新定义 getSalary 方法覆盖(override)超类中的 getSalary 方法。

```
class Manager extends Employee {
    . . .
    public double getSalary() { . . . }
    . . .
}
```

覆盖方法的实现

在 Manager 类中，getSalary 方法应当这样实现：

```
public double getSalary() {
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```

注意：

1. 不能直接访问超类中的 salary 域：

return salary + bonus;

会导致编译错误。因为 salary 是超类的私有字段。

2. 不能使用子类的 getSalary 方法获取超类的 salary 字段：

double baseSalary = getSalary();

这样会导致无限次循环调用自己。

3. super 是一个指示编译器调用超类方法的特有关键字。调用超类的公有方法和和公有字段时，必须使用该关键字指明。

在 JDK5.0 中，则放宽了要求：新方法的返回类型可以是原方法的返回类型的子类型。

例如：

假设在类 A 中有一个返回类型的 Employee 的方法 f；

假设 B 是 A 的子类，如果要在 B 中覆盖 f，则在 B 类中定义的 f 可以是：

```
Employee f(...) { ..... }
Manager f(...) { ..... }
```

注意：

1. 覆盖方法和原方法的签名(signature)必须相同。方法的签名包括方法的名字和参数列表。
2. 覆盖方法的可见性不能低于原方法的可见性。特别地，如果超类方法是 public，则子类方法必须也是 public。

super 关键字在构造器中的应用

Manager 类的构造器实现如下：

```
public Manager(String n, double s, int year, int month, int day) {
    super(n, s, year, month, day);
    bonus = 0;
}
```

关键字 super 的含义：是“调用超类 Employee 中含有 n, s, year, month 和 day 参数的构造器”。(由于 Manager 类不能访问 Employee 类的私有域，所以必须利用 Employee 类的构造方法对这部分私有域进行初始化。)

注意：1. 如果子类的构造器没有显式地调用超类的构造器，则会自动调用默认(没有参数)的构造器；

2.如果超类中没有不带参数的构造器，而子类的构造器中又没有显式地调用其它构造器，Java 编译器将报告错误。

super 与 this 的比较

this 关键字有两个用途：

1. 引用隐式参数
2. 调用该类其它的构造器

```
public Employee(double aSalary) {
    this( “Emplyoee #” + this.nextId, aSalary);
}
```

同样 super 关键字也有两个用途：

1. 调用超类的方法
2. 调用超类的构造器

注意：1. 在调用构造器时， super 与 this 的使用方式很相似：

super(...); // this(...)

2. 调用构造器的语句只能作为另一个构造器的第一条语句出现。

多态

多态源自置换法则：在程序中出现超类对象的任何地方都可以用子类对象置换。

例如，可以将一个子类的对象赋给超类变量：

```
Employee e;
```

e = new Employee(. . .); // e 被定义为 Employee 对象
e = new Manager(. . .); // e 指向 Manager 对象也可以

在 Java 中，对象变量是多态的，一个 Employee 类型变量既可以引用 Employee 类型对象，也可以引用一个 Employee 类的任何一个子类的对象(例如 Manager)。

注意：不能将一个超类的引用赋予子类变量：
Employee e;
Manager m = e; // ERROR

举例：ManagerTest
给出一个例子，实现如下功能：
1. 创建一个新经理对象
Manager boss = new Manager("Bill", 80000, 1987, 12, 15);
boss.setBonus(5000);

2. 定义一个 3 个雇员的数组：
Employee[] staff = new Employee[3];

3. 将经理和雇员放置于数组中：
staff[0] = boss;
staff[1] = new Employee("Harry", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony", 40000, 1990, 3, 15);
4. 输出每个人的薪水：
for (Employee e : staff)
    System.out.println(e.getName() + " " + e.getSalary());
输出结果如下：
Bill 85000.0
Harry 50000.0
Tony 40000.0

```
staff[0] = boss; //Manager对象
staff[1] = new Employee("Harry", …);
staff[2] = new Employee("Tony", …);
```

e.getSalary()能确定执行哪个 getSalary 方法(子类的？超类的？)。

方法调用过程

下面是对象方法调用过程的详细描述(动态绑定)：
1. 编译器查看对象的声明类型和方法名。假设调用 obj.f(param)，且 obj 被声明为 C 类的对象。
编译器会一一列举 C 类中所有名为 f 的所有方法及其超类中名为 f 而且是子类可访问的方法。
——至此，编译器已经获得所有可能被调用的方法。
2. 编译器将寻找与方法签名(signature)与 f 匹配的方法，这个过程成为重载解析(overloading resolution)。(如果 C 类的方法与超类签名相同，则 C 类方法覆盖超类方法)

——至此，编译器已获得需要调用的方法的名字和参数类型。

3.如果方法是 private、static、final 或者构造器，则编译器可以准确知道应该调用哪个方法。这种调用方式称为静态绑定(static binding)。

4. 调用哪个方法将依赖于隐式参数的类型，并在运行时动态绑定。虚拟机一定调用与变量所引用对象的实际类型最合适的那个类的方法。

例：for (Employee *e* : staff)

System.out.println(e.getName() + " " + e.getSalary());

(1)*e* 被声明为 Employee 类型，该类中只有一个名叫 getSalary 的方法，该方法没有参数，故不存在重载解析的问题。

(2)getSalary 方法不是 private 方法、static 方法或 final 方法，故将采用动态绑定。

(3)虚拟机为 Employee 和 Manager 两个类生成方法表。

Employee:	Manager:
getName()->Employee.getName();	getName()->Employee.getName();
getSalary()->Employee.getSalary();	getSalary()->Manager.getSalary();
getHireDay()->Employee.getHireDay();	getHireDay()->Employee.getHireDay();
raiseSalary(d)->Employee.raiseSalary(d);	raiseSalary(d)->Employee.raiseSalary(d);
	setBonus(d)->Manager.setBonus(d)

(4)虚拟机提取 *e* 的实际类型的方法表，并搜索定义 getSalary 签名的类，确定调用哪个方法。

动态绑定

- 执行过程中根据引用变量实际引用的对象类型调用相应的方法。
- 优点：无需对现存代码进行修改就可对程序进行扩展

```
Manager boss = new Executive("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
Employee[] staff = new Employee[2];
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
for (Employee e : staff)
    System.out.println("name=" + e.getName( )+ ",salary=" + e.getSalary());
```

```
public  void method( Employee e)
{
    e.getSalary( );
    …
}
```

阻止继承：final 类和 final 方法

有时候希望禁止某个类派生出子类。

final 类：不允许扩展的类。只需要在定义该类时加上 final 修饰符。

final 方法：将一个方法声明为 final 之后，子类就不能覆盖这个方法。(final 类中的所有方法自动地声明为 final 方法，但该类中的域不会自动声明为 final)

意义：阻止继承后，能确保 final 类和方法不会改变语意，即使是在子类中。

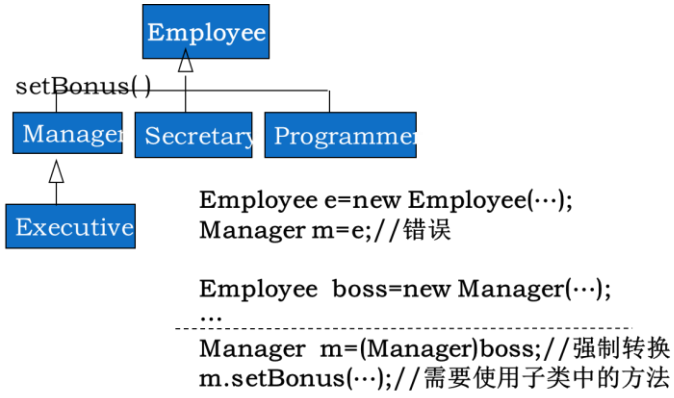
final 类和方法举例

```
public final Executive extends Manager {
    …
}
```

Executive 类已经是所有行政职务的最高层，完全有理由将其确定为 final

```
class Employee
{
    public final String getName()
    {…}
}
```

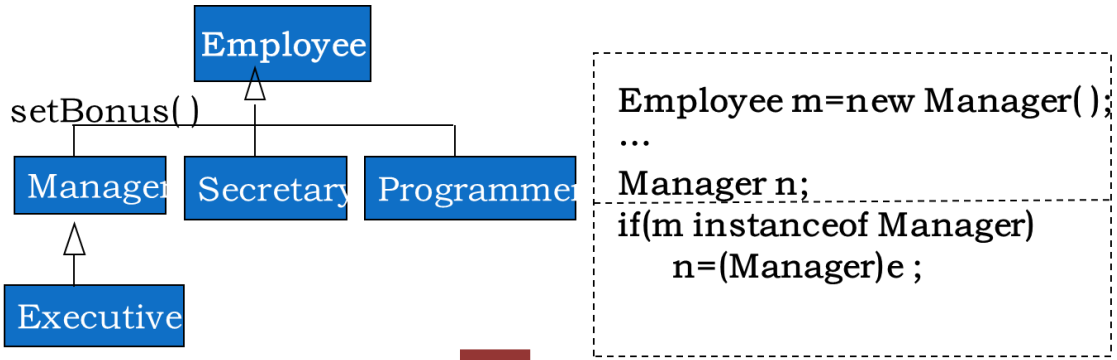
● 父类引用变量赋值给子类引用变量（强制转换）



● instanceof ： 引用变量 instanceof 类名

判断对象是否是某一类或该类的子类的对象

在强制转换之前一般使用 instanceof 检查转换是否正确



面向对象高级程序设计

抽象方法与抽象类

子类继承父类时,子类可选择是否重写父类的方法?是否可以强制要求子类重写某些方法?

可以.采用抽象方法和抽象类来强制子类重写某些方法.

抽象方法

抽象方法只需提供方法的声明,不用实现,即只有方法头,没有方法体,以关键字 abstract 标识;

定义语法:

[public|private...] *abstract* returnType abstractMethodName([paramlist])

示例:

public abstract void introduction();

抽象类

抽象类即是不能实例化的类;以关键字 abstract 标识:

定义语法:

[public|private...] *abstract* class abstractClassName

{

...

}

示例:

```
public abstract class Person{
    ……
    }
```

抽象类不一定要包含抽象方法;

抽象类可以有自己的构造方法和成员变量;

若类中包含了抽象方法, 则该类必须被定义为抽象类;

采用抽象类的一个原因是为了强制编程人员继承该类,同时实现该类中的抽象方法;

如果类 Student 继承自抽象类 Person,但并没有完全实现 Person 中的抽象方法,则类 Student 是否是抽象类?

不能创建抽象类的对象

可以定义抽象类的对象变量,但其只能引用非抽象子类的对象

示例:

假设 Person 是抽象类,Student 是其非抽象子类

```
Person p;
p=new Person( );//wrong
p=new Studnent( );//OK
```

域和方法的访问控制

变量和方法的访问控制符有：private、默认、protected 及 public。它们的具体访问范围如下：

访问指示	类	包	子类	所有
private	√			
默认(无修饰符)	√	√		
protected	√	√	√	
public	√	√	√	√

object 类

所有类都是 Object 的子类;

Object obj=new BankAccount();

Object obj=new String("Hello");

Object 类方法:

equals 方法

toString 方法

clone 方法

在自己编写的类中，最好重写这些方法!

equals 方法

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

判断两个对象是否具有相同的引用

在自己写的类中重写 equals 方法

```
class BankAccount{
    ……
public boolean equals(Object obj) {
```

```
  if(this==obj) return true;
  if(obj==null) return false;
  if(getClass()!=obj.getClass()) return false;

      BankAccount other=(BankAccount)obj;
    return (accountNumber==other.accountNumber &&
              balance==other.balance)
    }
}
```

子类中实现 equals 方法

首先调用超类的 equals，如果未通过测试，对象就不可能相等；

如果超类中的域都相等，就比较子类中的实例域；

```
class SavingAccount extends BankAccount{
      ……
   public boolean equals(Object obj){
          if(!super.equals(obj)) return false;

     SavingAccount other=(SaveingAccount)obj;
      return ……//比较 SavingAccount 中的成员变量
      }
}
```

Java 语言规范要求 equals 方法具有下面的特性

1)自反性:对于任何非空引用 x,x.equals(x)应该返回 true;

2)对称性:对于任何引用 x 和 y,如果 x.equals(y)返回 true,那么 y.equals(x)也应该返回 true;

3)传递性:对于任何引用 x,y,z,如果 x.equals(y)返回 true,y.equals(z)返回 true,那么 x.equals(z)也应该返回 true;

4)一致性:如果 x 和 y 引用的对象没有发生变化,那么反复调用 x.equals(y)应该返回同样的结果;

5)对于任意非空引用 x,x.equals(null)应该返回 false;

PPT37-38 的两个例子

注意:

如果子类能够拥有自己的相等概念,那么对称性需求将强制采用 getClass 进行检测;

如果由超类决定相等的概念,则可以使用 instanceof 进行检测,这样可以在不同子类的对象之间进行相等的比较.在超类中将方法设为 final.

hashCode 方法

hashCode 方法返回一个整型数值,并合理地组合实例域的散列码,以便能够让各个不同的对象产生的散列码更加均匀;

如果 x.equals(y)返回 true,那么 x.hashCode()就必须与 y.hashCode()具有相同的值;
如果重新定义 equals 方法,就必须重新定义 hashCode 方法;
Object 类的 hashCode()方法返回对象的存储地址;

```
class BankAccount{
……
public int hashCode(){
      return 7*Integer.hashCode(accountNumber)+11*Double().hashCode(balance);
      }
}
```

toString 方法

返回对象的字符串表示;
Object 类中默认返回对象所属的类名和散列码.

System.out.println(System.out);
输出:java.io.PrintStream@a90653

重写 toString 方法一般遵循的格式为:
类名[成员变量值]
java.awt.Point[x=20,y=20]
BankAccout 中的 toString 函数可改写为:

```
public String toString(){
        retrun “BankAccout[balance=“+balance+”,accountNumber=“+accountNumber+”]”;
      }
```

泛型程序设计

泛型程序设计(generic programming)：编写的代码能被多种不同类型的对象所重用。

Java 的泛型就是创建一个用类型作为参数的类。
就像我们写类的方法一样，方法是这样写的:
method(String str1,String str2)
方法中参数 str1、str2 的值是可变的。

而泛型也是一样的，这样写:
class Java_Generics<K, V>
其中 K 和 V 是类型变量，像方法中的参数 str1 和 str2 一样,也是可变的。

泛型数组列表

在 Java 中，ArrayList 类（java.util 包中）不需要在编译时就确定整个数组的大小。该类具备数组的基本特征，但在添加或者删除元素时，具有自动调节数组容量的功能。

在 JDK 1.5 中，ArrayList 是一个采用类型参数(type parameter)的泛型类(generic class)。举例：
ArrayList<Employee> staff = new ArrayList<Employee>();
上面的代码声明和构造了大小不定的 Employee 对象的数组。
只能使用 get 和 set 方法实现对数组列表元素的操作。例如：

```
ArrayList<Employee> staff = new ArrayList<Employee>();
……
staff.set(i, harry);
……
Employee e = staff.get(i);
```

ArrayList 类的重要方法

ArrayList<T> ():
构造一个空数组列表

ArrayList<T> (int initialCapacity):
用指定容量构造一个空数组列表

boolean add(T obj):
在数组列表末端添加一个元素

int size():
返回存储在数组列表中的当前元素数量

void trimToSize()
将数组列表存储容量削减为当前尺寸

Object[] toArray()
将数组列表转换为 Object 类型的数组

T get(int index):
返回指定位置的元素

void set(int index, T obj):
设置指定位置的元素值，这个操作将覆盖原有的值

void add(int index, T obj):
向后移动元素以便在指定位置插入元素

T remove(int index)
删除指定位置的元素，并将后面的元素向前移动，返回被删元素

对象包装器和自动装箱

对象包装器：将简单类型数据包装为复杂对象。
常见对象包装器：Integer, Long, Float, Double, Short, Byte, Character, Void, Boolean
对象包装器对象是不可变对象
对象包装器类是 final 类

```
ArrayList<int>  a ;
ArrayList<Integer>  list= new ArrayList<Integer>();
```

自动装箱

```
  list.add(3);
  list.add(new Integer(3));
```

自动拆箱

```
  int d = list.get(i);
  int d = list.get(i).intValue();

  Integer n = 3;
  n++;

  Integer a = 1000;
  Integer b = 1000;
  if(a==b){ ……}

  if(a.equals(b)){…}
```

Integer 的重要方法

```
int intValue()
static String toString(int i)
static String toString(int i, int radix)
static parseInt(String s)
static parseInt(String s, int radix)
static Integer valueOf(String s)
static Integer valueOf(String s, int radix)
```

参数数量可变

变参：可变的参数数量

```
  System.out.printf("%d", n);
```

System.out.printf("%d %s", n, "widgets");

public class PrintStream
{ public PrintStream printf(String fmt, Object... args) {...}}
——参数当中的"…"是 Java 代码的一部分，表明这个方法可以接收任意数量的对象；

用户可自定义可变参数的方法，并将参数指定为任意类型。

```
public static double max(double... values)
{
    double largest = Double.MIN_VALUE;
    for(double v:values) if (v > largest) largest = v;
    return largest;
}

double m = max(3.1, 40.4, -5);
```

枚举类

变量的取值范围在一个有限的集合内，可自定义枚举类型

```
enum Size{SMALL, MEDIUM, LARGE, EXTRA_LARGE};
Size s = Size.MEDIUM;
```

注：该声明定义的是类型是一个类，它刚好有 4 个实例，一般不会构造新对象。故在比较两个枚举类型的值时，不需要调用 equals，直接使用"=="就行。

可在枚举类型中添加一些构造器、方法和域。

```
public enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L");
    private String abbreviation;
    private Size(String abbreviation)
      {this.abbreviation = abbreviation;}
    public String getAbbreviation(){return abbreviation;}
}
```

Enum 类

所有枚举类型都是 Enum 的子类，它们继承了这个类的许多方法。

主要方法

static Enum valueOf(Class enumClass, String name)

返回指定名字、给定类的枚举常量。

Size s = Enum.valueOf(Size.class, "SMALL");

Size s = Size.SMALL;
String toString()
返回枚举常量名
int ordinal( )
返回枚举常量在 enum 声明中的位置，从 0 开始计数。
Int compareTo(E other)
根据枚举常量在 enum 声明中的位置，返回负值、0、正值。

继承设计的技巧

将公共操作和域放在超类；
不要使用受保护的域；
使用继承实现"is-a"关系；继承设计的技巧

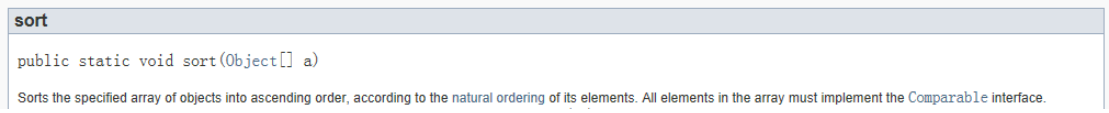
除非所有继承的方法都有意义，否则不要使用继承；
在覆盖方法时，不要改变预期的行为；
使用多态，而非类型信息。

接口和内部类

接口

接口不是类，而是对类的一组需求描述，这些类要遵从接口描述的统一格式进行定义。
接口可用来规范类的行为，只包含常量和方法的定义，而没有变量和方法的实现

Eg: Arrays 的 sort 方法可对对象进行排序。



All elements in the array must implement the Comparable interface.

排序的前提：数组的元素对象所属的类实现了 Comparable 接口。

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

接口的定义

接口的定义包括接口声明和接口体
完整的接口声明：
[public] interface interfaceName[extends listOfSuperInterface] { … }
接口体包括常量定义和方法定义

-常量定义格式为：type NAME=value；该常量被实现该接口的多个类共享;缺省的具有 public ,final, static 的属性。

-方法定义格式为：(缺省的具有 public 和 abstract 属性)

returnType methodName([paramlist]);

Example:

```
public interface Comparable<T>
{
   int compareTo(T other);
}
```

接口的实现

在类的声明中用 implements 子句来表示一个类实现某个接口

一个类可以实现多个接口,在 implements 子句中用逗号分开

在类体中可以使用接口中定义的常量，必须实现接口中定义的所有方法。

例:

```
public class Employee implements Comparable<Employee>
{
    // Other Employee methods
    public int compareTo(Employee other)
    {
        return Double.compare(salary, other.salary);
    }
}
```

接口类型的使用

接口作为一种引用类型来使用

可以声明接口的一个变量，但不能实例化

```
Comparable m=new Comparable(); //wrong
```

接口可以指向任何实现该接口的类的实例；

```
Comparable m=new Employee();
```

通过这些接口变量可以访问类所实现的接口中的方法。

接口和类的互换

```
public interface Measurable
{
    double getMeasure();
```

```
}
class BankAccount implements Measurable…
class Student implements Measurable…
class DataSet{
   …
   public void add(Measurable x){…};
   public Measurable getMaxium(){…};
}
DataSet bankData=new DataSet();
    bankData.add(new BankAccount(1100));

    public void add(Measurable x)
```

如果一个类实现了某个接口，则将该类转换为该接口是正确的，而且不需要强制转换；

接口转换为某个实现了该接口的类，需要用强制转换

```
    BankAccount m=(BankAccout)bankData.getMaxium();
      m.deposit(3000);
```

接口与多态

接口不能实例化

```
     Measurable  m=new Measuable();  //wrong!
```

一个对象只能是实现了该接口的某个类的实例

```
     Measurable x;
     x = new BankAccount(10000);
     x = new Studnet( “Jack");
```

调用对象的方法

```
    x.getMeasure();
```

接口可实现运行时的多态，在运行阶段，由对象的实际类型决定调用哪一个具体的方法

-如果 x 是一个 BankAccount 对象，则调用 BankAccount.getMeasure();

-如果 x 是一个 Student 对象，则调用 Student.getMeasure();

接口 VS 抽象类

接口中所有的方法都是抽象的，即有方法名，方法参数，返回值，但没有方法体；抽象类中可有具体方法；

接口中的方法的访问限制权限都是 public;抽象类中无此限制；

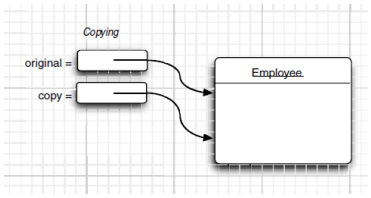
接口中没有成员变量，但可以有常量；抽象类中可有成员变量；

一个类只能继承一个抽象类，但可实现多个接口。

对象克隆

拷贝

```
Employee original = new Employee(...);
Employee copy = original;
copy.raiseSalary(10);
```



克隆

```
Employee copy = original.clone();
copy.raiseSalary(10);
```

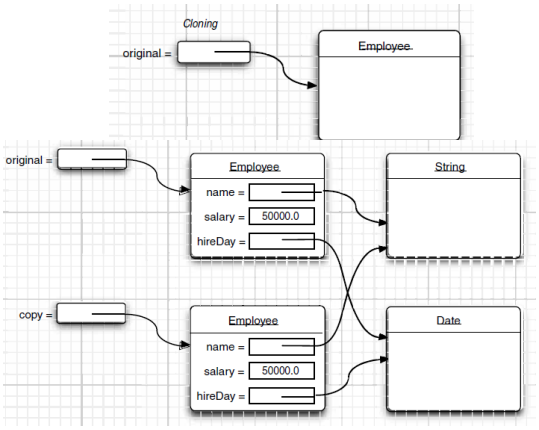
Object: clone -------浅拷贝

protected Object clone()

throws CloneNotSupportedException{...}

Object 中的克隆是浅拷贝

如果类中实例域中不包含可变对象，则浅拷贝可满足需求，否则需要重写 clone 方法



重写 clone 方法

在以下情况下需重写 clone 方法：

1.类中实例域含有可变对象；

2.即使类中实例域只包含有简单类型成员和不可变对象，但允许其它类克隆本类对象；

类重写 clone 方法，必须：

1 实现 Cloneable 接口；

2 使用访问修饰符 public 重新定义 clone 方法；

不包含可变实例域

```
class Employee implements Cloneable
{
    public Employee clone()
        throws CloneNotSupportedException
    {
        return (Employee) super.clone();
    }
}
```

包含可变实例域

```
class Employee implements Cloneable
{
    public Employee clone()
        throws CloneNotSupportedException
```

```
        {
            Employee cloned = (Employee)super.clone();
            cloned.hireDay = (Date) hireDay.clone();
            return cloned;
        }
    }
```

接口与回调

回调(callback)：一种程序设计模式，可以指出某个事件发生时应该采取的动作。

例：Timer,构造一个定时器，每隔 10 秒钟打印一条信息"At the tone, the time is …"，并响一声

```
Timer(int interval, ActionListener listener)

public interface ActionListener
{
        void actionPerformed(ActionEvent event)
}
class TimePrinter implements ActionListener
{
        public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
            System.out.println("At the tone,…"+now);
            Toolkit.getDefaultToolkit().beep();
        }
}
TimePrinter listener = new TimePrinter();
Timer t = new Timer(10000, listener);
t.start();
```

内部类

内部类：定义在另一个类中的类。

内部类的作用：

1 内部类方法可以访问该类定义所在的作用域中的数据，包括私有的数据；

2 内部类可以对同一个包中的其他类隐藏起来；

3 匿名内部类可用于定义一个回调函数且不想编写大量代码时。

使用内部类访问对象状态

例：语音时钟类 TalkingClock

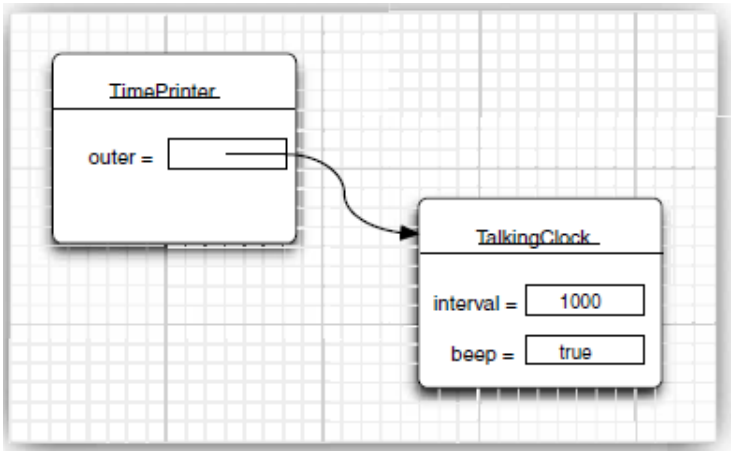
```
class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep){...}
    public void start() {...}

    public class TimePrinter implements ActionListener
    {
      ...
     }
 }
public class TimePrinter implements ActionListener
  {
        public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
            System.out.println("...")+now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
  }
```

内部类对象既可以访问自身的数据域，也可以访问创建它的外围对象的数据域。

内部类的对象总有一个隐式引用，它指向了创建它的外围类对象。



该引用不可见，为说明这种方式，将内部类对外围类对象的引用称为 outer。

```
public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
```

```
            System.out.println("..."+now);
            if (outer.beep) Toolkit.getDefaultToolkit().beep();
        }
```

内部类中对外围类的引用在构造器中设置。编译器修改了所有的内部类的构造器，增加一个外围类的引用参数。

```
 public TimePrinter(TalkingClock clock)
{
        outer = clock;
 }
```

start 方法中创建对象 TimePrinter 后，编译器会将 this 引用传递给当前的语音时钟的构造器：

```
 ActionListener listener = new TimePrinter(this);
```

内部类的特殊语法规则

外围类引用的正规语法

Outer←-->OuterClass.this

```
 public void actionPerformed(ActionEvent event)
{
    Date now = new Date();
   System.out.println("..."+now);
    if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

内部类对象构造器调用正规语法

OuterObject.new InnerClass(paras)

Eg:

```
 ActionListener listener = this.new TimePrinter();

TalkingClock jab = new TalkingClock(1000, true);
TalkingClock.TimePrinter lis = jab.new TimePrinter();
```

内部类的编译

编译器会把内部类翻译成用$分隔外部类名与内部类名的常规类文件。

TalkingClock$TimePrinter.class

局部内部类/局部类

局部类：在方法当中定义的内部类；

局部类不能用 public 或 private 访问说明符进行声明，它的作用域被限定在声明这个局部类的块中；

局部类可对外部世界完全隐藏；

例：

```
public void start()
{
    class TimePrinter implements ActionListener
   {
          public void actionPerformed(ActionEvent event)
          {...
          }
    }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(interval, listener);
   t.start();
}
```

由外部方法访问 final 变量

局部类不仅能够访问包含它们的外部类，还可以访问局部变量，但局部变量必须被声明为 final：

```
public void start(int interval, final boolean beep)
{
    class TimePrinter implements ActionListener
   {
          public void actionPerformed(ActionEvent event)
          { ...
            if(beep) Toolkit.getDefaultToolkit().beep();
          }
    }
   ActionListener listener = new TimePrinter();
   Timer t = new Timer(interval, listener);
   t.start();
}
```

Why final?

程序控制流程：

1 调用 start 方法；

2 调用内部类 TimePrinter 的构造器，以便初始化对象 listener 变量；

3 将 listener 引用传递给 Timer 构造器，定时器开始计时，start 方法结束，start 方法的参数变量

beep 不复存在；

4 然后，actionPerformed 方法执行 if(beep)…。

TimePrinter 类需在域 beep 释放之前将 beep 域保存，故需要用 final 。

匿名内部类

```
public void start(int interval, final boolean beep)
{
  ActionListener listener = new ActionListener()
 {
     public void actionPerformed(ActionEvent event)
          { …
          }
   };
  Timer t = new Timer(interval, listener);
  t.start();
}
```

语法：

```
    new SuperType(construction parameters)
    {
                inner class methods and data
   }

  new InterfaceType( )
    {
       method and data
     }
```

静态内部类

把一个类隐藏在另外一个类的内部，并不需要内部类引用外围类对象，可用静态内部类。

Java 的异常处理

异常就是在程序的执行过程中所发生的异常事件，它中断指令的正常执行

异常

可能出现的问题

用户输入错误

设备错误
物理限制

异常处理的目标

向用户通报错误，返回到一种安全状态，并能够让用户执行一些其他的命令；
允许用户保存所有操作的结果，并以适当的方式终止程序。

异常处理的方法

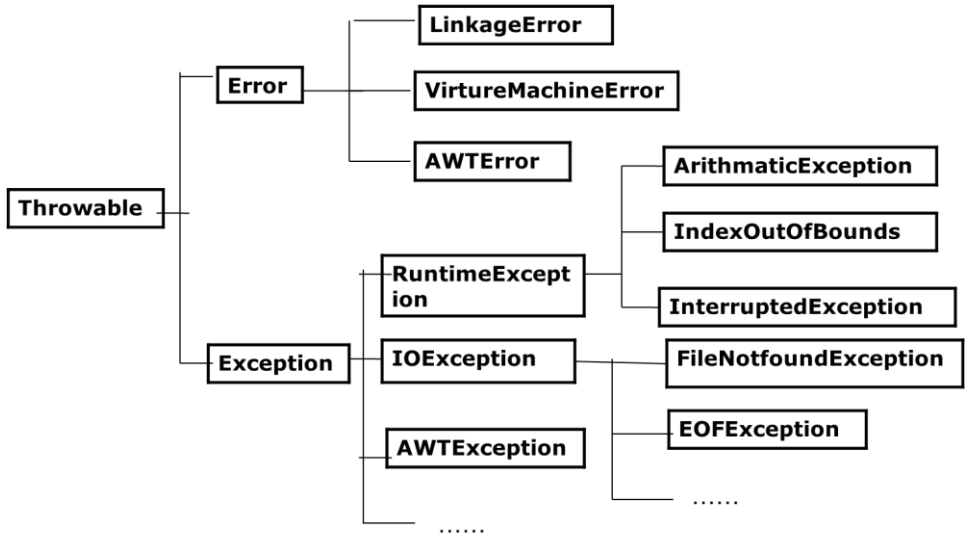
“如果某个方法不能够采用正常的途径完成任务，就可以通过另外一个途径退出方法”
•产生异常的方法立刻退出，且不返回任何值，而是抛出了一个封装了错误信息的对象；
•调用产生异常方法的代码也将无法继续执行，而是，异常处理机制开始搜索能够处理这种异常状况的异常处理器。

异常（Throwable）分类

Throwable:所有异常的根类
Error:Throwable 的直接子类
动态链接失败，虚拟机错误等。通常 Java 程序不应该捕获这类异常，也不会抛弃这种异常。
Exception
•运行时异常
继承于 RuntimeException.Java
一般是由于程序错误产生
-错误的类型转换
-数组访问越界
-访问空指针
……
编译器允许不对它们做出处理(unchecked)
“如果出现 RuntimeException 异常，就一定是你的问题”

•非运行时异常
除了运行时异常之外的其它的继承自 Exception 的异常类。
程序曾经能够正常运行，但由于某种情况的变化，导致异常出现，程序不能正常运行。
-文件不存在
-用户输入错误
……

编译器要求程序必须对这类异常进行处理(checked)



异常处理机制

在 java 的执行过程中，如果出现了异常事件，就会生成一个异常对象。JAVA 有两种对异常的处理方式：

•抛弃异常：方法内部产生异常的地方，生成一个异常对象，并将该异常对象提交给方法的调用者，这一异常的生成和提交过称成为抛弃（throw）异常

•捕获异常：当 Java 运行时得到一个异常对象时，它将会寻找处理这一异常的代码。找到能够处理这种类型的异常的方法后，系统把当前异常对象交给这个方法处理，这一过程成为捕获（catch）异常

抛弃异常

声明抛弃异常

•声明抛弃异常是在一个方法声明中的 throws 子句中指明的。例如

```
public FileInputStream(String name )throws FileNotFoundException
```

•throws 子句中可以同时指明多个异常，说明该方法将不对这些异常进行处理，而是声明抛弃它们。

```
public static void main(String args[])throws IOException,IndexOutOfBoundsException
```

如何抛弃异常

•抛弃异常首先要生成异常对象，异常对象可由某些类的实例生成，也可以由 JAVA 虚拟机生成。抛弃异常对象时通过 throw 语句实现。

```
IOException e=new IOException( );
throw e;
```

•可以抛弃的异常必须是 Throwable 或者其子类的实例。

下面的语句在编译时将会产生语法错误

```
throw new String("want to throw");
```

•对于已存在的异常类，抛出该异常非常容易
1.找到一个合适的异常类；
2.创建这个类的一个对象；
3.将该对象抛出；

•对于已存在的异常类，抛出该异常非常容易

```
String readData(Scanner in)throws EOFException
 {
      …
      while(…)
      {   if(!in.hasNext( ))
          {
               if(n<len)
                    throw new EOFException( );
          }
      }
}
```

自定义异常类

自定义异常类即是定义一个派生于 Exception 的直接或间接子类；如一个派生于 IOException 的类。

一般情况下，定义的异常类应该包括两个构造器，一个是默认的构造器，一个是带有详细描述信息的构造器（超类 Throwable 的 toString 方法会打印出这些详细信息，有利调试代码）

```
自定义类
class FileFormatException extends IOException{

    public FileFormatException(){}
     public FileFormatException(String gripe)
     {
          super(gripe);
     }

}
String readData(BufferedReader in)throws FileFormatException{
      ……
      while(…)
      {
          if((ch=read())==-1)
          {
               if(n<len)
                    throw new FileFormatException(“File
```

```
                    format error!");
                }
            }
        }
```

捕获异常

捕获异常是通过 try-catch-finally 语句实现的

```
try{
    ……
   }catch(ExceptionName1 e){
    ……
   } catch(ExceptionName2 e){
   ……
   }finally{
   ……
}
```

• try

捕获异常的第一步使用 try{……}选定捕获异常的范围，由 try 所限定的代码块中的语句在执行过程中可能会生成异常对象

• catch

-每个 try 代码块可以伴随一个或多个 catch 语句，用于处理 try 代码块中所生成的异常事件。

-catch 语句只需要一个形式参数指明它所能捕获的异常类型，这个类必须是 Throwable 的子类，运行时系统通过参数值把被抛弃的异常对象传寄给 catch 块

-在 catch 块中是对异常对象进行处理的代码，与访问其他对象一样，可以访问一个异常对象的变量或调用它的方法。getMessage（）是类 Throwable 所提供的方法，用来得到有关异常事件的信息，类 Throwable 还提供了方法 printStackTrace（）用来跟踪异常事件发生时执行堆栈的内容

```
try{
……
}catch（FileNotFoundException e）{
  System.out.println(e);
  System.out.println("message:"+e.getMessage());
  e.printStackTrace(System.out);
}catch(IOException e){
 System.out.println(e);
}
```

catch 语句的顺序

捕获异常的顺序和不同 catch 语句的顺序有关，当捕获到一个异常时。剩下的 catch 语句就不再进行匹配。

因此在安排 catch 语句的顺序时，首先应该捕获最特殊的异常，然后在逐渐一般化，也就是一般先安排子类，再安排父类。

- **finally**

捕获异常的最后一步是通过 finally 语句为异常处理提供一个统一的出口，使得控制流程转到程序的其它部分以前，能够对程序的状态做统一的管理。不论在 try 代码块中是否发生了异常事件，finally 块中的语句都会被执行。

示例：

```
try{
        //1
      code that might throw exceptions
      //2
    }
catch(IOException e)
   {
     //3
     show error dialog
     }
finally{
      //4
}
//5
……
```

程序没有产生异常，1-2-4-5

程序产生了一个可被 catch 捕获的异常：1-3-4-5

程序产生了一个不能被 catch 捕获的异常：1-4

异常处理

积极处理和消极处理

处理原则

捕获那些知道如何处理的异常；

将不知道如何处理的异常抛出；

使用异常机制的建议

（1）异常不能代替简单的测试：只在异常情况下使用异常机制

例：上百万地对一个空栈进行退栈操作

```
if(!s.empty()) s.pop();
```

```
try{
      s.pop();
}
catch(EmptyStackException e)
{
}
```

Test	Throw/Catch
154 milliseconds	10739milliseconds

（2）不要过分细化异常，尽量将有可能产生异常的语句放在一个 try 语句块中；

（3）抛出的异常类型尽可能明确，不要总是抛出 IOException；

（4）早抛出，晚捕获，尽量让高层次的方法通告用户发生了错误；

Java 图形程序设计

Swing 概述

两种基本 GUI 程序设计类库

AWT(Abstract Window Toolkit)抽象窗口工具箱

-对等体方法

将处理用户界面元素的任务委派给每个目标平台的本地 GUI 工具箱，由本地 GUI 工具箱负责用户界面元素的创建和动作。

SWING

-1996，Netscape 创建了一种 IFC(Internet Foundation Class)的 GUI 库，它将菜单、按钮等用户界面元素绘制在空白窗口上，而对等体只需创建和绘制窗口。

在所有平台上的外观和动作都一样。

-Sun 和 Netscape 合作完善了这种方式，创建了一个名为 Swing 的用户界面库。

SWING vs. AWT

-SWING 显示用户界面的元素的速度比 AWT 慢一些；

-SWING 拥有一个丰富、便捷的用户界面元素集合；

-SWING 对低层平台的依赖很少，因此与平台相关的 bug 很少；

-SWING 给予不同平台的用户一致的感观效果；

•SWING 没有完全替代 AWT，而是基于 AWT 架构之上，其提供了能力更加强大的用户界面组件，但还需要使用基本的 AWT 事件处理。

GUI 基本组成

Java 的图形用户界面的最基本组成成分是组件，组件是一个可以以图形化的方式显示在屏幕上并能与用户进行交互的对象，例如一个按钮，一个标签等。

组件不能独立地显示出来，必须将组件放在一定的容器中才可以显示出来。

容器（Container）实际上是 Component 的子类，因此容器本身也是一个组件，具有组件的所有性质，另外还具有容纳其他组件和容器的功能

创建框架

框架(frame)

顶层窗口被称为框架

Swing 用 JFrame 类来表示框架，该 类扩展于 AWT 的 frame。

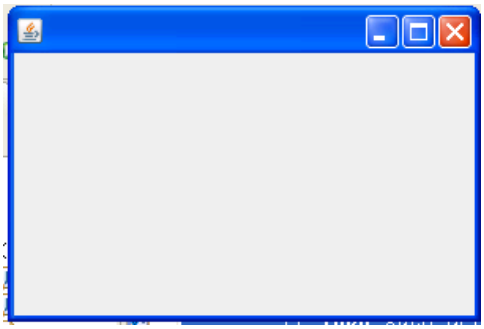
JFrame 是极少数几个不绘制在画布上的 Swing 组件之一。其修饰部件（按钮、标题栏、图标等）由用户的窗口系统绘制，而不是由 Swing 绘制。

JFrame

```
java.long.Object
    |
    +----java.awt.Component
              |
         +----java.awt.Container
              |
              +----java.awt.Window
                        |
                   +----java.awt.Frame
                             |
                        +----javax.swing.JFrame
```

```
import javax.swing.*;
import java.awt.*;
public class SimpleFrameTest {
    public static void main(String[] args) {
      EventQuene.involkLater(() ->
      {
```

```
        SimpleFrame  sFrame = new SimpleFrame();
        sFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        sFrame.setVisible(true);
         })
    }
}
//所有的 swing 组件必须由事件分派线程进行配置，将鼠标点击和按键控制转移到用户接口组件。
class SimpleFrame extends JFrame{
        public SimpleFrame(){
                setSize(DEFAULT_WIDTH,DEFAULT_HEIGHT);
        }
        public static int DEFAULT_WIDTH = 300;
        public static int DEFAULT_HEIGHT = 200;
}
```



默认情况下,框架的大小为 0*0 象素;
默认情况下,用户关闭窗口只是将框架隐藏了起来,程序并没有终止;
构造一个框架并不自动显示,框架起初并不可见;

设置合适的框架大小

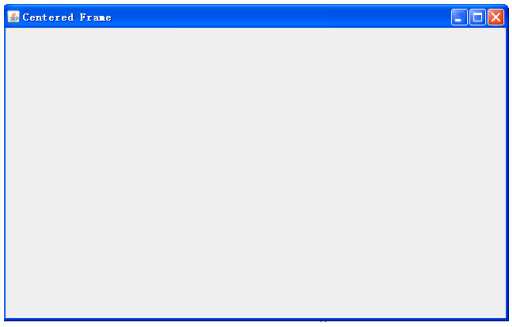
-获得用户系统的基于像素的屏幕分辨率信息,然后利用这些信息计算最佳的窗口大小。
-获得用户系统的屏幕分辨率信息

```
        Toolkit tk = Toolkit.getDefaultToolkit();
        Dimension ds = tk.getScreenSize();
        int width = ds.width;
        int hight = ds.height;
```

例：将一个可关闭框架设置为：①其大小是整个屏幕的四分之一；②位于屏幕的中央。
void setLocation(x,y);
将框架放置在左上角水平 x 像素，垂直 y 像素的位置；坐标(0,0）位于屏幕的左上角；
void setTitle(String s);
设置框架的标题;
void setIconImage(Image c);
设置框架的图标;

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CenteredFrameTest {
      public static void main(String[]args){
        EventQuene.involkLater(() ->
        {
              CenteredFrame cf = new CenteredFrame();
               cf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               cf.setVisible(true);
         }
      }
}
class CenteredFrame extends JFrame{
      public CenteredFrame(){
            Toolkit tk = Toolkit.getDefaultToolkit();
            Dimension ds = tk.getScreenSize();
            int width = ds.width;
            int hight = ds.height;

            setSize(width/2,hight/2);
            setLocation(width/4,hight/4);
            Image img = tk.getImage("icon.gif");
            setIconImage(img);
            setTitle("Centered Frame");
      }
}
```



在框架中显示文本信息

可以直接在框架中绘制信息,但这不是一种良好的编程习惯;

框架在 JAVA 中是一个容器，用来放置其它组件，如：放置一个按钮、一个文本框等;

通常在一个组件上绘制信息，并将该组件添加到框架中;