第5章 深入理解JAVA语言

部分内容摘自

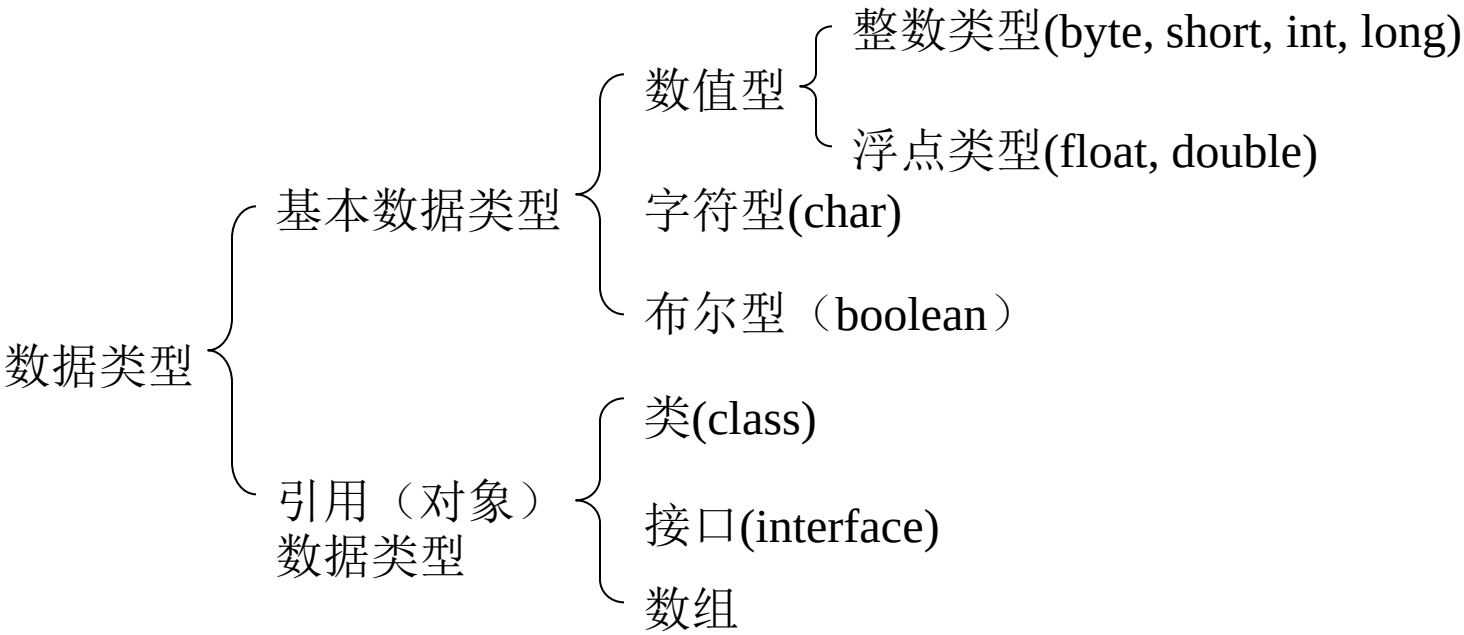
《Java面向对象编程》，孙卫琴

《Java 程序设计》，唐大仕

5.1 变量及其传递

Java数据类型划分

- **Java**中的数据类型分为两大类，一类是基本数据类型（**primitive types**），
- 另一类是引用类型**(reference types)**。后者相当于对象。



ch05.ref_val.ObjectAndPrimitive

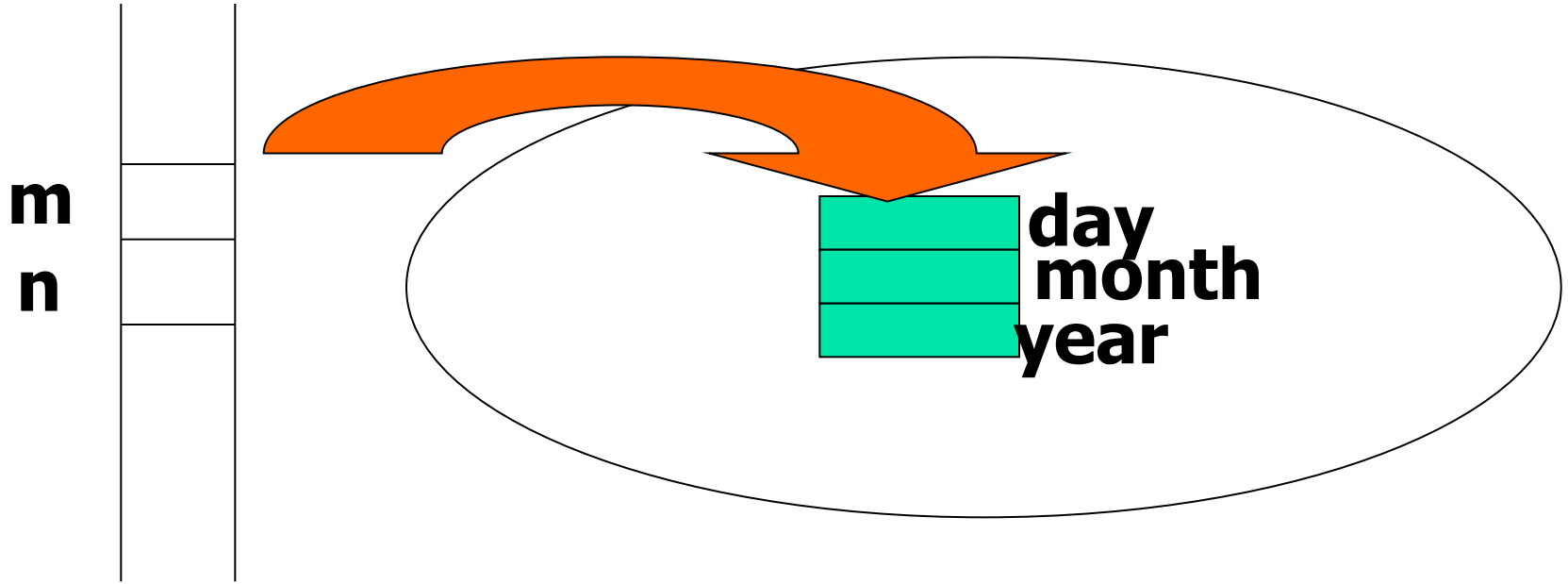
从JDK1.4以后，引用/对象类型和原始类型基本通用

类型	引用/对象类型	原始数据类型
整数	Integer	int
单精度	Float	float
双精度	Double	double
布尔	Boolean	boolean
字符	Character	char
字节	Byte	byte

5.1 变量及其传递

- 基本类型变量与引用型变量
- 基本类型：其值直接存于变量中。
- 引用型的变量除占据一定的内存空间外，它所引用的对象实体（由**new** 创建）也要占据一定空间。
- 引用型变量保存的实际上是对象在内存的地址，也称为对象的句柄。
- **ch05.ref_val.MyDate** 体会**m/n**什么时候一起改变，什么时候不一起改变。

引用型变量与对象实体的关系- ch05.ref_val.MyDate



5.1 变量的传递

- 调用对象方法时，要传递参数。在传递参数时，**Java** 是值传递，即在调用一个方法时，是将表达式的值复制给形式参数。
- 对于引用型变量，传递的值是引用值(可以理解为内存地址)。
- **ch05.ref_val.TransByValue.java**

```
public class TransByValue {
  private static int a;
  public static void main(String[] args) {
    int a=0;
    modify(a);
    System.out.println(a);

    int[] b=new int[1];
    modify(b);
    System.out.println(b[0]); //1 or 5
}
  public static void modify(int a){
     a++;
   }
  public static void modify(int[] b){
     b[0]++;
     b=new int[5];
 }
} ///:~
```

- **Java**中的参数都是按值传递的，但对于引用型变量，传递的值是引用值，所以方法中对数据的操作可以改变对象的属性。
- 但是不能简单的认为函数如果传递对象的是对象，就可以在函数内部修改这个参数，例如：ch05.ref_val. TestValue

实例变量与局部变量

- 从语法角度看
  - 实例变量属于类或接口；**public,private,static,final** 修饰。
  - 而局部变量是在方法中定义的变量或方法的参变量。
  - 都可用**final**修饰，但局部变量则不能够被访问控制符及**static**修饰。
- 从存储角度看
  - 从变量在内存中的存储方式来看，实例变量是对象的一部分，而对象是存在于堆中的，局部变量是存在于栈中。
  - 实例变量的生命周期与局部变量的生命周期比较。
  - 另外，实例变量可以自动赋初值，局部变量则须显式赋值。局部变量必须显示赋值后才能够使用。

实例变量/局部变量 **ch05.** LocalVarAndMemberVar

```
class Test()
{
   int a;//默认有初始值
   void m(){
      int b;//默认没有初始值
      System.out.println(b);//编译不能通过需要
//初始化。
   }
}
```

变量的返回

- 方法的返回：
  - 返回基本类型。
  - 返回引用类型。它就可以存取对象实体。
- **Object getNewObject()**
- **{**
- **Object obj=new Object();**
- **return obj;**
- **}**

- 调用时：**Object p= GetNewObject();**

5.2 多态

- 子类对象可以当做父类对象，对于重载的方法，**Java**运行时根据实际的类型调用正确的方法，就叫“多态型性”。
- 所有的非**final/static/private**方法都可以实现多态。
- 多态的例子：**ch05.virtual.TestVirtualInvoke.java** 非常重要
- 多态能够使对象所编写的程序，不用做修改就可以适应于其所有的子类，如在调用方法时，程序会正确地调用子对象的方法。例如：**ch05.intergate.IntegrationDemo**
- 多态的特点大大提高了程序的抽象程度和简洁性，最大限度地降低了类和程序模块之间的耦合性，提高了类模块的封闭性，使得它们不需了解对方的具体细节，就可以很好地共同工作。这个优点对于程序的设计、开发和维护都有很大的好处。

5.3 Class类

- 对象可以通过**getClass()**方法来获得运行时的信息。
- **getClass()**是**java.lang.Object**的方法，而**Object**是所有类的父类。所以任何对象都可以用**getClass()**方法。
- 例子**ch05.RunTimeClassInfo.java**
- **getClass()**方法得到对象的运行时的类信息，即一个**Class**类的对象，它的**getFields()**及**getMethods()**方法能进一步获得其详细信息

instanceof 一个对象是否是某个类/接口（或子类，或实现了这个接口）

用法：对象名 instanceof 类名

```
String str="abc";
if(str instanceof String){}//true
Object obj =str;
if(obj instanceof String){} //true
obj =new Object();
if(obj instanceof String){} //false
Frog frog=new Frog();
if(frog instanceof swimmer){} //true
```

5.3 对象构造与初始化

- 调用本类或父类的构造方法
- **this**调用本类的其他构造方法。
- **super**调用直接父类的构造方法
- 如果没有**this**及**super**，则编译器自动加上**super()**，即调用直接父类不带参数的构造方法。
- **this**或**super**要放在第一条语句,且只能够有一条.

```
Class A
{
  A(int a){}
}

Class B extends A
{
  B(String s){} //编译不能够通过.
}
```

- 编译器会自动调用**B(String s){ super();}** 出错.
- 解决方法:
  - 在**B**的构造方法中,加入**super(3);**
  - 在**A**中加入一个不带参数的构造方法,**A(){}**
  - 去掉**A**中全部的构造方法,则编译器会自动加入一个不带参数的构造方法,称为默认的构造方法.

例子：ch05.seq.ConstructCallThisAndSuper

- **在构造函数中使用this和super.**
- 在构造方法中调用this及super或自动加入的super，最终保证了任何一个构造方法都要调用父类的构造方法，而父类的构造方法又会再调用其父类的构造方法，直到最顶层的Object类。这是符合面向对象的概念的，因为必须令所有父类的构造方法都得到调用，否则整个对象的构建就可能不正确。

构造方法的执行过程

对于一个复杂的对象，构造方法的执行过程遵照以下步骤：

- 调用本类或父类的构造方法，直至最深一层派生类。
- 按照声明顺序执行域的初始化赋值。
- 执行构造函数中的各语句。
- **ch05.seq.ConstructSequence.java**
- 构建器的调用顺序非常重要。先父类构造，再本类成员赋值，最后执行构造方法中的语句。

构造方法内部调用的方法的多态性

- 在构造子类的一个对象时，子类构造方法会调用父类的构造方法，而如果父类的构造方法中调用到对象的其他方法，如果所调用的方法又被子类所覆盖的话，它可能实际上调用的是子类的方法，这是由动态绑定（虚方法调用）所决定的。从语法上来说这是合理的，但有时会造成事实上的不合理，所以在构造方法中调用其他方法要小心。
- ch05.metamorph.**ConstructInvoke.java**

5.4 对象清除与垃圾回收

- **new**创建对象。
- 自动清除，清除过程称为垃圾回收。
- 对象回收是由 **Java**虚拟机的垃圾回收线程来完成的。
- 系统中的任何对象都有一个引用计数器，当其值为**0**时，说明该对象可以回收。
- 垃圾回收：**System.gc()**方法 。它可以要求系统进行垃圾回收。但它仅仅只有建议权。

5.4finalize()方法

- **Object**具有**finalize()**方法，类**C++**中析构函数。
- 可以通过覆盖**Object** 的**finalize()**方法来实现关闭打开的文件、清除一些非内存资源等工作需要在对象懂得回收时进行。因为系统在回收时会自动调用对象的**finalize()** 方法。
- 一般来说，子类的**finalize()**方法中应该调用父类的**finalize()**方法，以保证父类的清理工作能够正常进行。
- 但是程序绝对不能完全依赖**finalize()**释放资源，因为**finalize()**是不可控的。
- 例如： ch05.clean. TestFinalize
- 为了准确释放资源，应该用**try finally**或者 AutoCloseable 接口(这个地方不清楚以后会详细讲)

5.5 内部类与匿名类

- 内部类是在其他类中的类。
- 匿名类是一种特殊的内部类，它没有类名，在定义类的同时就生成该对象的一个实例。

5.5.1内部类的定义和使用

- 将类的定义置入一个用于封装它的类内部即可。
- 内部类不能够与外部类同名。
- 在封装它的类的内部使用内部类，与普通类的使用方式相同，在其他地方使用，类名前要冠以外部类的名字。在用**new**创建内部类时，也要在 **new**前面冠以对象变量。
- ch05.inner.**InnerUse.java**

5.5.2. 在内部类中使用外部类的成员

- 内部类中可以直接访问外部类的其他域及方法。即使**private**也行。
- 如内部类中有与外部类同名的域或方法，可以用**this**来访问外部成员。
- ch05.inner.**TestInnerThis.java**

5.5.3.内部类的修饰符

- 内部类与类中的域、方法一样是外部类的成员，它的前面也可以有访问控制符和其他修饰符。内部类可用的修饰符比外部类的修饰符更多。（外部类不能够使用 **protected,private,static** 等修饰，而内部类可以。**)**
- 访问控制符：**public,protected,**默认及 **private,final,abstract**。
- 用**static**修饰表明该内部类实际是一种外部类。

5.5.4 static内部类 ch05.inner.TestInnerStatic

- **static** 环境在使用时要遵循以下规则：
- **1**、实例化**static**内部类时，在 **new**前面不需要用对象变量；
- **2**、**static**内部类中不能访问其外部类的非**static**的域及方法，既只能够访问**static**成员。
- **3**、**static**方法中不能访问非**static**的域及方法，也不能够不带前缀地**new** 一个非**static**的内部类。

5.5.5 方法中的内部类及匿名类 一般了解

5.5.6方法中的内部类

- 在一个方法中也可以定义类。这种类称为方法中的 内部类。
- 例子：ch05.inner.**TestInnerMethod**

- **1**、同局部变量一样，方法中的内部类前不能够用**public,private,protected,static**修饰，但可以被**final**或者**abstract**修饰。
- **2**、方法中的内部类可以访问其外部类的成员；若是**Static**中的内部类可以访问外部类的**static**成员。
- **3**、方法中的内部类中，不能够访问该方法的局部变量，除非是**final**局部变量。
- **4**、方法中定义的类，在其他地方使用时，没有类的情况，正像例中一样，只能够用其父类来引用这样的变量。

匿名类ch05.inner.TestInnerAnonymous

- 匿名类有以下几个特点：
- **1**、不取名字，直接用其父类的名字。
- **2**、类的定义域创建该类的一个实例同时进行，即类的定义前面有一个**new**。不使用关键词**class**。

  new 类名或接口名（）**{……}**。
- **3**、类名前面不能够有修饰符。
- **4**、类中不能够定义构造方法，因为它没有名字。在构造对象时不能够带参数。

lambda表达式、注解、反射课程后期讲