对象及内存管理

了解Java都知道，Java内存管理中存在垃圾回收机制，也就是对于那些在内存中，失去引用的对象会被回收掉，以清空内存。因此造成大多数人肆无忌惮的挥霍内存。但往往这样得不偿失，会造成程序运行效率低下。本文总结来自于李刚对《Java程序员的基本修养》。Never say you know Java!!!

- Java内存管理
- 实例变量和类变量
- 构造器
- 父类与子类
- 关键字final

知识点扫盲

1. Java内存管理
   Java内存管理分两个方面，一是内存分配，一是内存回收。内存分配指的是创建Java对象时，为这个对象在堆中分配的内存；内存回收指的是在某个Java对象失去引用时，JVM的垃圾回收机制会对其进行回收，以清空内存。
2. 实例变量和类变量
   当我们定义一个类的时候，总有些变量会包含在这个类的定义中，这些变量称为成员变量(Field)。如果该成员变量被关键字static修饰，那么我们称为该成员变量为静态变量(类变量)；否则称为非静态变量(实例变量)。类变量是在类加载时，就加载到堆内存中的，对于每一个类，类变量只有一份，所有类的实例共享这一份变量；而实例变量是每个实例对象所拥有的，有多少个实例对象就有多少份实例变量。
3. 实例变量的初始化 实例变量的初始化方式有三种：
   - 为实例变量指定初始值
   - 利用初始化块为实例变量赋值
   - 利用构造器为实例变量赋值

   值得注意的是，前两种方法的执行时间要先于第三种方法的执行时间，且他们在编译器里面执行的顺序和他们在源代码中的顺序一致。因此如果指定赋值初始化和初始化代码块初始化有重复赋值，那么后一种将会覆盖前一种。
   看下面例子：

```
class Test {
int a;
int b;
//构造器初始化
public Test(int a, int b) {
    this.a = a;
    this.b = b;
}
double d = 4.0; //指定赋值初始化
{
    //初始化代码块初始化
    d = 9.0;//覆盖指定赋值初始化的4.0
    c = 3.0;
}
double c = 2.0;//指定赋值初始化，覆盖初始化代码块中3.0
}
```

4. 类变量的初始化
   因为类变量是直属于类的，所以它不存在用构造器来进行初始化，所以类变量的初始化有两种方式：
   - 指定赋值初始化
   - 静态赋值代码块初始化

   同样的，他们俩在编译器里面执行的顺序和他们在源码中的顺序是一致的，同样会存在覆盖问题。
   请看例子：

```
class Test {
static String name = "test1";  //指定赋值初始化
static {
    //静态复制代码块初始化
    name = "test2";//覆盖指定赋值初始化
    age = 30;
}
static int age = 20; //覆盖静态代码块初始化
}
```

来看一个有趣的例子，这个例子也是来自于基本修养这本书，在Eclipse测验过，很说明问题。

```
 public class PriceTest {
 public static void main(String[] args) {
     System.out.println(Price.INSTANCE.currentPrice);
     Price price = new Price(2.8);
     System.out.println(price.currentPrice);
 }
}
class Price {
 static double initPrice = 20;//注释这条，结果不同，一个为-2.8，一个为17.2

 final static Price INSTANCE = new Price(2.8);

 //static double initPrice = 20;//注释这条，输出的结果相同，都为17.2

 double currentPrice;

 public Price(double discount) {
     currentPrice = initPrice - discount;
 }
}
```

在Price类中，我们定义了两个类变量，其中一个还是Price的实例对象。从上面代码中，发现为什么注释不同的语句，会产生不同的结果。仔细分析这份代码，类变量的初始化基本上就没问题了。 首先JVM加载Price类，会在堆中分配两份内存给类变量，一份为INSTANCE，一份为initPrice,初始值分别为null和0.0。然后按照类变量的初始化方式为这两个类变量进行初始化：

- 若 `static double initPrice = 20;` 这条语句在 `final static Price INSTANCE = new Price(2.8);` 之后，那么当在堆中new Price对象是，initPrice还是系统默认赋值(因为是按照源码中的初始化顺序来进行初始化的)，也就是0.0。所以对于INSTANCE对象，其currentPrice是 `0.0-2.8 = -2.8`；
- 若 `static double initPrice = 20;` 这条语句在 `final static Price INSTANCE = new Price(2.8);` 之前，那么当在堆中new Price对象是，initPrice已经是指定赋值初始化的20了。所以对于INSTANCE对象，其currentPrice是 `20.0-2.8 = 17.2`；
- 而对于新new出来的对象，那时候initPrice早已经初始化成功，因此输出来的结果是17.2。

5. 构造器

    - 5.1: 父类构造器的调用
      在Java中，创建某一个类的实例对象时，系统总是先调用最顶层父类的初始化操作，包括初始化块代码和构造器，然后逐层调用初始化块代码和构造器直至本类。但是在调用父类构造器的过程中，可能存在多个构造器，调用父类哪个构造器，这是由子类构造器和父类构造器中的this关键字决定的。

所以总的来说，子类构造器决定调用父类的哪一个构造器，但是调用过程是从最顶层父类到最底层本类的。
那么子类是按照什么规则调用父类构造器的呢？分下面三种情况：

- 子类构造器的第一行代码使用super关键字来**显示**调用父类构造器，并且根据super的参数来决定调用父类的特定构造器
- 子类构造器的第一行代码使用this关键字来**显示**调用本类其他构造器，至于调用哪个构造器，由传入this方法的参数决定
- 如果子类构造器的第一行代码既没有显示调用父类构造器，也没有显示调用本类其余构造器，那么就**隐式**调用上层父类无参构造器

> *值得注意的是，super()方法和this()方法必须出现在子类构造器的第一行代码，且super()是显示调用父类构造器，this()是显示调用本类其余构造器。在每个构造器中，super()和this()只能使用其中一个，而且只能调用一次，且必须在构造器中第一行代码。*

- 5.2: 父类和子类的变量和方法访问
  - 父类可以访问子类对象的实例变量吗？
    通常情况下，而且是绝大多数情况下，父类是不能访问子类的实例变量的，因为父类不知道是哪个子类继承了自己，而且可能还在此基础上添加了一些实例变量。但是子类是可以访问父类实例变量的，因为子类继承父类就会获得父类的变量和方法。但在某些极端情况下，父类是可以访问子类的实例变量的，例子可以参考基本素养这本书P34-P37.
  - 父类可以调用子类的方法吗？
    绝大多数情况下，父类是不能调用子类的方法的，因为父类不知道哪个子类会继承自己，而且还可能在此基础上添加一些新的方法。但是子类是可以调用父类的方法的，因为子类继承父类就继承了它的方法和实例变量。但有一种特殊情况，即当子类重写父类的某个方法时，表面上可能是父类在调用自己的方法，其实是在调用子类方法。
    由于我们上面提到，当调用子类构造器时，如果不指定super和this等显示方法，系统会隐式的调用父类的无参构造方法，如果此时在父类的无参构造方法中调用了父类被子类重写的方法，那么此时调用的就是子类的方法，而非父类的方法；而且如果在子类的这个重写的方法中使用了子类构造器中的一些初始值，那么此时就会出错，因为这个方法的调用是在初始化之前执行的。***因此千万不要在父类构造器中调用可能被子类重写的方法。***
- 5.3: 在继承中内存控制
  继承是OOP三大特征之一，其余两个是封装和多态。现在仔细看看在继承中，父子类的内存控制。这部分在看书过程中会觉得枯燥难懂，我尽量用简洁语言总结。
  - 在继子类继承父类的过程中，对于父类的实例变量和方法的继承方式是不同的。对于父类方法的继承，子类是完全继承，且如果子类有重写父类的方

法，那么子类重写的方法会覆盖父类的方法；但是对于父类的实例变量来说，就算子类中也存在和父类实例变量名一致的实例变量，子类实例变量是不会覆盖父类的实例变量的，而是在子类实例化的过程中，父类的实例变量被隐藏起来了。***因为在继承过程中，对于父类的实例变量和方法存在这样的继承差别，因此对于一个引用变量而言，当我们通过这个变量去访问实例变量时，我们实际上访问的是声明该变量类型的那个类所含有的实例变量；而当我们通过这个应用变量去调用方法时，我们实际上调用的是这个变量实际真正上引用的对象的方法。***

这样说也许有些抽象，用一个小例子来举例说明:

```
    public class Test {
    public static void main(String[] args) {
    Base base = new Ext();
    System.out.println(base.count);
    base.display();
    }
}
class Base {
    int count = 2;
    public void display() {
        System.out.println(this.count);
    }
}
class Ext extends Base {
    int count = 20;
    @Override
    public void display() {
        System.out.println(this.count);
    }
}
```

在这个例子中，Base是父类，Ext是继承父类的子类，父类和子类中都包含了一个实例变量叫做count，和一个方法叫做display()，只不过我们在子类中重写了这个父类函数。在测试程序中，我们声明了一个Base类的引用变量，他指向的是在内存中一块是Ext的实例对象。当我们直接调用base.count时，就像我们所说的那样，实际上访问的是声明该变量所包含的那个实例变量，而如果调用display()方法时，实际上调用的是在内存中真正存在的那个实例对象，也就是Ext对象。所以测试程序输出结果为2，20.

- 内存中子类的存放
  在上面我们提到当子类继承父类之后，对于父类中的实例变量并不会进行覆盖，而是进行隐藏。因此在实例化子类的过程中，只会存在一个对象，但是在这个对象中包含了他所继承点所有父类的实例变量，注意，并不存在父类对象在内存中。但是如何访问父类的实例变量呢？通常我们是不需要这么做的，但是万一要访问，怎么实现？就像上面提到的那样，只要将引用变量声明为父类变量就好了。

看下面例子:

```
  public class Test {
  public static void main(String[] args) {
  Sub sub = new Sub();
  Mid mid = sub;
  Base base = sub;
  System.out.println(sub.count);
  System.out.println(mid.count);
  System.out.println(base.count);
  sub.info();
  mid.info();
  base.info();
}
}
class Base {
int count = 2;
public void info() {
  System.out.println("base");
}
}
class Mid extends Base {
int count = 20;
@Override
public void info() {
  System.out.println("mid");
}
}
class Sub extends Mid {
int count = 200;
@Override
public void info() {
  System.out.println("sub");
}
}
```

在这个例子中，我们首先在内存中开辟了一个Sub对象，然后再往上转型，赋值给Mid,和Sub。此时如果访问实例变量count，会出现不同的结果，因此这个测试程序输出结果为200，20，2.但是如果在父类中有方法在子类中被重写了，无论是哪种情况下都会调用最低层那个方法。在这个测试程序中输出的都为“sub”。

- 父子类的类变量 类变量不像实例变量那么复杂。因为它是直属于类本身，而并非某个对象实例。因此可以直接用类名来访问类变量。甭管它是否是在子类中调用的。

6. final修饰符

记得我们在数组中提到过，对于基本数据类型和引用数据类型，如果程序员不显示地初始

化，那么系统会自动的初始化这些变量。但是对于用final修饰符修饰的数据类型是个特例，它必须由程序员显示的初始化值。被final修饰符修饰的变量，一旦被初始化之后，就再也不能被赋值。
由final修饰符修饰的实例变量有三种初始化方法:

- 在定义时指定初始化赋值
- 在非静态初始化代码块中进行初始化
- 在构造器中进行初始化
  看下面例子:

```
class Test {
  final int a1 = 2; //指定赋值初始化
  final int a2;
  final int a3;
  {
      //在非静态代码块中初始化
      a2 = 3;
  }
  public Test() {
      //构造器中初始化
      this.a3 = 4;
  }
}
```

由final修饰符修饰的类变量有两种初始化方法:

- 在定义时指定赋值初始化
- 在静态初始化代码块中初始化

看下面例子:

```
class Test {
  final static int a1 = 2; //指定赋值初始化
  final static int a2;
  static{
      //在静态代码块中初始化
      a2 = 3;
  }
  }
```

宏变量
final修饰符一个重要的作用就是宏变量替换。被final修饰符修饰的变量，不管实例变量还是类变量，都会在JVM加载编译类是就已经初始化好了，意味着这一切发生在所有操作之前，包括类变量的初始化之前。
记住一点，在编译阶段就已经确定好对变量就可以叫做宏变量

几种会在编译阶段确定好的表达式：

- 当然被final修饰符修饰的变量的***直接赋值***情况会在编译阶段确定(在静态／非静态代码块以及构造器中初始化的final修饰变量也不会进行宏替换，即不会在编译阶段确定)；
- 如果被赋的表达式只是基本的算术表达式或者字符串连接，不涉及到变量访问，就会在编译阶段确定。

看个小例子：

```
public class Test {
    public static void main(String[] args) {
        String s1 = "Crazy Java";
        String s2 = "Crazy " + "Java";
        System.out.println(s1 == s2);
        String str1 = "Crazy ";
        String str2 = "Java";
        String s3 = str1 + str2;
        System.out.println(s1 == s3);
    }
}
```

在这个例子中，输出的结果为true，false。其原因就是s1和s2是简单的赋值和字符串连接，并没有涉及到变量访问，因此字编译的时候，系统会在string pool中找到这个“Crazy Java”字符串，然后赋给s1和s2.但是对于s3来说，它是由两个字符串访问连接得到的，因此不能在编译阶段确定，而且根据字符串的不可变性，字符串的连接其实会在内存开辟新的空间，因此s3和s1在内存中是分属于不同地址。

内部类的局部变量

在Java中，局部内部类包括匿名内部类是可以访问局部变量的，其他的普通内部类和普通静态内部类是不能访问局部变量的。被局部内部类或者匿名内部类访问的局部变量必须用final修饰符修饰。作为结论记住，有兴趣的可以去搜搜Java中的“隐式闭包”现象。