目 录

1 第一个 Java 程序

1.1 名字管理

Java 为了给一个类库生成不会与其他名字混淆的名字，让程序员反过来使用自己的 Internet 域名，从而保证它们是独一无二的。比如域名为 MindView.net，那么各种应用工具库就被命名为 net.mindview.utility.foibles。反转域名后，句点就用来代表子目录的划分。

1.2 static 关键字

现在有两个需求，如下所示：

- 只想为某个特定域分配单一存储空间，而不去考虑究竟要创建多少对象，甚至根本就不创建任何对象。
- 希望某个函数不与包含它的类的任何对象关联在一起。也就是说，即使没有创建对象，也可以调用这个函数。

static 关键字可以满足这两方面的需求。当声明一个事物是 static 的时候，就意味着这个域或这个函数不会与包含它的那个类的任何对象实例关联在一起。

引用 static 变量有两种方法。可以通过一个对象去定位它，也可以通过其类名直接引用。

static 函数的重要用法是在不创建任何对象的前提下就可以调用它。Java 中禁止使用全局函数，这时候 static 就派上了用场。程序员可以通过类本身来调用 static 函数和 static 变量。在没有全局变量和全局函数的情况下，可以在类中置入 static 函数和 static 域。在其他类中就可以直接通过类名访问这些 static 函数和 static 域。

1.3 Hello world

下面是第一个完整的程序，可以打印出“hello world”：

```
1  import java.util.*;
2
3  public class HelloWorld
4  {
5      public static void main(String[] args)
6      {
7          System.out.println("hello world");
8      }
9  }
```

1.4 编码风格

代码风格的规定如下：

- 类名的首字母要大些。如果类名由几个单词构成，那么就把它们并在一起，其中每个内部单词的首字母都采用大写形式。
- 其他内容，第一个字母采用小写。如果该内容由几个单词构成，那么就把它们并在一起，其中每个内部单词的首字母都采用大写形式。

2 初始化与清理

2.1 初始化

类似于 C++ 的方式，Java 也采用了构造器进行初始化。例子如下：

```
class Rock
{
    Rock()
    {
        System.out.println("Hello world");
    }
}
```

和 C++ 一样，Java 的构造器也可以进行重载。例子如下：

```
class Tree
{
    int height;
    Tree()
    {
        System.out.println("Planting a seeding");
        height = 0;
    }
    Tree(int inititalHeight)
    {
        height = inititalHeight;
        System.out.println("Creating new Tree that is " + height + " feer tall"
            );
    }
}
```

关于默认构造器，这里需要注意：如果程序员没有提供任何构造器，编译器将自动生成一个默认构造器；如果程序员写了一个构造器，编译器就不会自动生成一个默认构造器。

2.2 this 关键字

Java 的 this 关键字和 C++ 的 this 关键字有区别。C++ 的 this 关键字是该对象的地址，而 Java 中是没有指针的。Java 的 this 关键字表示该对象的引用。例子如下：

```
public class Leaf
{
    int i = 0;
    Leaf increment()
    {
        ++i;
        return this; // 相当于返回当前对象
    }
```

```
9     }
```

2.2.1 利用 this 调用构造器

在构造器中，如果为 this 添加了参数列表，将产生对符合此参数列表的某个构造器的明确调用。例子如下：

```
1   public class Flower
2   {
3       int petalCount = 0;
4       String s = "initial value";
5       Flower(int petals)
6       {
7           petalCount = petals;
8       }
9       Flower(String ss)
10      {
11          s = ss;
12      }
13      Flower(String s, int petals)
14      {
15          this(petals);
16          // 不能再写this(s)，否则会覆盖原有的内容
17          this.s = s;
18      }
19      void print()
20      {
21          // 在非构造函数中不能使用this(s)
22          System.out.println("Hello world!");
23      }
24  }
```

2.3 成员初始化

即使程序员没有对类的每个基本类型数据成员初始化，它们都会有一个初始值。在类中定义一个对象引用时，如果不将其初始化，该对象引用的初值就是 null。

与 C++ 不同的是，如果想为类中某个变量赋初值，只要在定义类成员变量的地方将其赋值即可，而 C++ 不允许这样的行为。

2.3.1 使用构造器进行初始化

可以使用构造器进行初始化，但是这无法阻止自动初始化的进行，它将在构造器被调用前发生。

在类的内部，变量定义的先后顺序决定了初始化的顺序。即使变量定义散布于函数定义之间，它们也将在任何函数 (包括构造器) 被调用之前得到初始化。例子如下：

```
class Window
{
    Window(int order)
    {
        System.out.println("order is " + order);
    }
}

class House
{
    Window w1 = new Window(1);
    House()
    {
        w3 = new Window(4);
    }
    Window w2 = new Window(2);
    Window w3 = new Window(3);
}

public class OrderOfInitialization
{
    public static void main(String[] args)
    {
        House h = new House();
    }
}
```

Java 中 static 关键字不能应用于局部变量，只能作用于域。当类中的静态基本类型域没有初始化时，它的值会是基本类型的标准初值。如果它是一个对象引用，那么它的默认初始化值是 null。如果想在定义处进行初始化静态数据，采取的方法与非晶态数据没有什么不同。

静态变量只有在必要时刻才会初始化。只有类被第一次访问或使用时，类中静态成员才会被初始化，而且是类中所有静态成员变量一起被初始化。此后，静态对象不会再次被初始化。需要注意的是，静态成员变量在非静态成员变量之前被初始化。例子如下:

```
class Bowl
{
    Bowl(int marker)
    {
        System.out.println("order is " + marker);
    }
    void f1()
    {
        System.out.println("f1: Hello world");
    }
}

class Table
{
    static Bowl bowl1 = new Bowl(1);
    Table()
    {
```

```
18              System.out.println("Table()");
19              bowl2.foo();
20          }
21          void foo2()
22          {
23              System.out.println("foo2: Hello world");
24          }
25          static Bowl bowl2 = new Bowl(2);
26      }
27
28      class CupBoard
29      {
30          Bowl bowl3 = new Bowl(3);
31          static Bowl bowl4 = new Bowl(4);
32          CupBoard()
33          {
34              System.out.println("CupBoard()");
35              bowl4.foo1();
36          }
37          void foo3()
38          {
39              System.out.println("foo3: Hello world");
40          }
41          static Bowl bowl5 = new Bowl(5);
42      }
43
44      public class StaticInitialization
45      {
46          public static void main(String[] args)
47          {
48              System.out.println("Creating new CupBoard() in main");
49              new CupBoard();
50              System.out.println("Creating new CupBoard() in main");
51              new CupBoard();
52              table.foo2();
53              cupboard.foo3();
54          }
55          static Table table = new Table();
56          static CupBoard cupboard = new CupBoard();
57      }
```

假设有一个名为 Dog 的类，Dog 对象创建的步骤如下：

- 当首次创建类型为 Dog 的对象，或者 Dog 类的静态成员变量或静态成员函数 (构造器本身也是静态函数) 被访问到时，Java 解释器必须查找类路径，用于定位 Dog.class 文件，然后载入 Dog.class。
- 载入 Dog.class 之后，将执行静态初始化的所有动作。也就是说，静态初始化在类对象首次加载时进行。
- 使用 new 创建对象时，会在堆上为 Dog 对象分配足够的存储空间。然后将这块存储空间清零，这样一来，Dog 对象中所有基本类型都被设置成了默认值，而引用被设置为 null。

- 随后执行所有出现于字段定义处的初始化动作。
- 执行构造器。

2.3.2 显式的静态初始化

Java 允许将多个静态初始化动作组织成一个特殊的“静态子句”。例子如下：

```
public class Spoon
{
    static int i;
    static
    {
        i = 47;
    }
}
```

静态子句只会执行一次。当首次生成这个类的一个对象或者首次访问属于这个类的静态成员变量或静态成员函数时，该代码段将执行。

2.3.3 数组初始化

Java 不允许指定数组的大小。例子如下：

```
// 现在拥有的是对数组的一个引用
// 我们并没有给数组对象本身分配任何空间
// 我们只是为该引用分配了空间
int a[];
```

2.4 清理

Java 拥有一个垃圾回收器，用于释放由 new 分配的内存。在其他情况下，如果该内存区域不是由 new 分配，那么垃圾回收器就不知道该如何释放这块特殊内存。

为了解决这种情况，Java 允许在类中定义一个名为 finalize() 的函数。一旦垃圾回收器准备释放对象占用的存储空间，将首先调用其 finalize() 函数，并在垃圾回收动作发生时，真正地回收对象占用的内存。

finalize() 函数与 C++ 中的析构函数有区别，因为 C++ 中的对象一定会被销毁，而 Java 中的对象不一定总是被垃圾回收器回收，而且垃圾回收并不等于析构。需要知道的是，只要程序没有濒临存储空间用完的时刻，垃圾回收器就不会释放程序所创建的任何对象的存储空间。

2.4.1 finalize 函数的用途

finalize 函数不会负责释放对象所占有的内存。无论对象是如何创建的，垃圾回收器都会负责释放对象占据的所有内存。只有当 Java 程序中调用了本地方法分配空间时，finalize 函数才会派上用场。本地方法是一种在 Java 中调用非 Java 代码的方式。在非 Java 代码中，可能会调用 C 的 malloc 函数分配存储空间，此时只有使用了 free 函数，该内存空间才会释放。所以在对象释放时，需要在 finalize 函数中调用 free 函数来释放这块特殊的内存，否则将引起内存泄漏。

下面是使用 finalize() 函数的例子：

```
class Book
{
    boolean checkedOut = false;
    Book(boolean checkOut)
    {
        checkedOut = checkOut;
    }
    void checkIn()
    {
        checkedOut = false;
    }
    protected void finalize()
    {
        if(checkedOut)
            System.out.println("Error: checked out");
    }
}

public class TerminationCondition
{
    public static void main(String[] args)
    {
        Book novel = new Book(true);
        novel.checkIn();
        new Book(true);
        System.gc();
    }
}
```

以上程序通过 finalize() 函数确保所有 Book 对象在被当作垃圾回收钱都应该被 check in。当有一本书没有 check in 时，程序将输出错误情况。类似的，如果一个对象代表了一个打开的文件，在对象被回收前程序员应该关闭这个文件。finalize() 函数可以用来检查文件是否关闭。finalize 函数更多地被用来发现程序中隐晦的缺陷。

2.4.2 垃圾回收器的工作机制

首先需要意识到，Java 中除了基本类型，所有对象都在堆上分配空间。然而 Java 从堆分配空间的速度，可以和其他语言从堆栈上分配空间的速度相媲美。

对比一下 Java 与 C++ 的堆空间分配机制。C++ 的堆像一个院子，里面的每个对象都

负责管理自己的地盘。如果对象被销毁了，这个地盘必须加以重新利用。在 Java 中，堆更像一个堆栈，每分配一个新对象，它就往前移动一格。

Java 这样的实现方式会导致频繁的内存页面调度。为了避免这种情况的出现，Java 使用了垃圾回收器。垃圾回收器工作的时候，一边回收空间，一边讲堆中的对象排列紧凑。通过垃圾回收器对对象重新排列，实现了一种告诉的、有无限空间可供分配的堆模型。

Java 垃圾回收器依据的思想是：对任何活的对象，一定能追溯到其存活在堆栈或静态存储区之中的引用。那么，只要从堆栈或静态存储区之中的引用开始遍历，就能找到所有活的对象。

Java 垃圾回收器处理存活对象的方式为：先暂停程序的运行，将所有存活的对象从当前堆复制到另一个堆，没有复制的全都是垃圾。当对象被复制到新堆，它们是紧凑排列的。这就避免了内存页面调度频繁的发生。这种做法称为“停止-复制”。

如果 Java 虚拟机发现程序很少产生垃圾甚至不产生垃圾时，Java 会切换到另一种工作模式，叫做“标记-清扫”。“标记-清扫”的思路是：从堆栈和静态存储区出发，遍历所有引用，进而找出所有存活的对象。每当它找到一个存活对象，就会给对象设一个标记。当标记完所有存活的对象后，清理工作才开始进行。在清理过程中，没有标记的对象会被释放。这样一来，就不会有复制动作的发生，但是剩下的堆空间也会不连续。垃圾回收器要是希望得到连续空间，就要重新整理剩下的对象。“标记-清扫”工作必须在程序暂停的情况下才能进行。

3 访问权限控制

Java 有四个访问权限：public、protected、包访问权限 (没有关键词) 和 private。

3.1 包: 库单元

包内包含一组类，它们在单一的名字空间之下被组织在一起。使用 package 语句指定类的名字空间。例子如下：

```
1  package psd.mypackage;
2
3  public class MyClass
4  {
5      // ...
6  }
```

当同一目录下其他文件想调用这个类时，需要使用 import 语句。例子如下：

```
1  import psd.mypackage.MyClass;
2
3  public class ImportedMyClass
4  {
5      public static void main(String[] args)
6      {
7          MyClass m = new MyClass();
8      }
9  }
```

如果两个文件不在同一目录下，就需要设置 CLASSPATH。Java 中的包名相当于路径名。当一个文件声明 package psd.mypackage 的时候，这个文件应该处于 CLASSPATH/psd/mypackage 的目录下。如果其他目录下的文件想要调用时，只有写 import psd.mypackage.*，那么这个文件就会搜寻 CLASSPATH/psd/mypackage 的目录下的所有文件。

3.2 Java 访问权限修饰词

3.2.1 包访问权限

当类或成员没有关键词时，对于同一包下的文件，对这个类或成员都有访问权限。而对于其他包的所有类，这个类或成员就是 private 的。

3.2.2 public 访问权限

当一个类被 public 修饰，那么就表明自己对其他人都是可用的。例子如下：

```
1  package access.dessert;
```

```
2
3   public class Cookie
4   {
5       public Cookie()
6       {
7           System.out.println("Hello world");
8       }
9       void bite()
10      {
11          System.out.println("bite");
12      }
13  }
```

Cookie 类是 public 的，所有其他文件可以使用它创建对象。需要注意的是，bite() 成员函数是具有包访问权限的，所以只有同一包的文件才能访问它。例子如下：

```
1   import access.dessert.*;
2
3   public class Dinner
4   {
5       public static void main(String[] args)
6       {
7           Cookie x = new Cookie();
8           // x.bite(); error!!!
9       }
10  }
```

3.2.3 private 访问权限

当一个成员被 private 修饰，那么这个成员只能在类中使用，和 C++ 的用法是一样的。例子如下：

```
1   class Sundae
2   {
3       private Sundae(){}
4       static Sundae makeASundae()
5       {
6           return new Sundae();
7       }
8   }
9
10  public class IceCream
11  {
12      public static void main(String[] args)
13      {
14          // Sundae x = new Sundae(); error!!!
15          Sundae x = Sundae.makeASundae();
16      }
17  }
```

3.2.4 protected 访问权限

protected 访问权限就是继承访问权限，和 C++ 的用法一样。例子如下：

```
package access.cookie2.*;

public class Cookie
{
    public Cookie()
    {
        System.out.println("Cookie constructor");
    }
    protected void bite()
    {
        System.out.println("bite");
    }
}
```

处于其他类继承这个类时，就可以访问 bite() 函数。例子如下：

```
import access.cookie2.Cookie;

public class Chocolate extends Cookie
{
    public Chocolate()
    {
        System.out.println("Chocolate constructor");
    }
    public void chomp()
    {
        bite();
    }
}
```

3.3 public 类

一个类只有两种访问权限：包访问权限和 public。只有一个类被修饰为 public，其他包下的类才可以创建该类的对象。与 public 类有关的限制如下：

- 每个文件中只能有一个 public 类。
- public 类的名称必须和文件名完全一样。

4 复用类

Java 中复用类的两种方式：

- 在新的类中产生现有类的对象，这种方法称为组合。
- 按照现有类的类型来创建新类，这种方法称为继承。

4.1 类的组合

组合技术很直观，只要将对象引用置于新类中即可。例子如下：

```
class WaterSource
{
    private String s;
    WaterSource()
    {
        System.out.println("WaterSource");
        s = "Constructed";
    }
}
```

4.2 类的继承

Java 中继承的语法和 C++ 类似，不过 Java 中使用关键字 extends 声明。如果继承基类，新类就会得到基类中所有非私有的域和成员函数。例子如下：

```
class Cleanser
{
    private String s = "Cleanser";
    public void append(String a)
    {
        s += a;
    }
    public static void main(String[] args)
    {
        Cleanser x = new Cleanser();
        x.append(" hello world");
    }
}

public class Detergent extends Cleanser
{
    public static void main(String[] args)
    {
        Detergent x = new Detergent();
        x.append(" hello world");
        Cleanser.main(args);
    }
```

```
23      }
```

4.2.1 基类的初始化

如果没有特别声明，将调用基类默认的构造器或者无参数构造器。如果想调用一个带参数的基类构造器，就必须使用 super 显式地调用基类构造器。例子如下：

```
1   class Game
2   {
3       Game(int i)
4       {
5           System.out.println("Hello World");
6       }
7   }
8
9   public class Chess extends Game
10  {
11      Chess()
12      {
13          super(1);
14          System.out.println("Chess constructor");
15      }
16      public static void main(String[] args)
17      {
18          Chess c = new Chess();
19      }
20  }
```

4.2.2 名称屏蔽

与 C++ 不同的是，Java 中导出类如果重载基类中的函数，并不会屏蔽其在基类中该函数的任何版本。例子如下：

```
1   class Homer
2   {
3       char doh(char c)
4       {
5           return c;
6       }
7       float doh(float c)
8       {
9           return c;
10      }
11  }
12
13  class Bart extends Homer
14  {
15      String doh(String s)
16      {
17          return s;
```

```
        }
    }
```

需要注意的是，因为这个语法特点，Java 中其实是没有名称屏蔽的。那么当我们要覆写基类中的一个函数时，很可能将其重载而非覆写。为了防止这个错误的发生，Java 提供了 @Override 注解相应的函数。如果这个函数是重载而非覆写时，编译器就会产生错误：

```
class Lisa extends Homer
{
    @Override
    String doh(String s)
    {
        return s; // 将产生错误
    }
}
```

4.2.3 向上转型

继承技术最重要的不是为新的类提供函数，而是用于表现新类和基类之间的关系。新类是现有类的一种类型。例子如下：

```
class Instrument
{
    public void play(){}
    static void tune(Instrument i)
    {
        i.play();
    }
}

public class Wind extends Instrument
{
    public static void main(String[] args)
    {
        Wind flute = new Wind();
        Instrument.tune(flute);
        // tune函数接受的是Instrument对象
        // 这里它也可以接受Wind对象
        // 因为Wind是Instrument的一种类型
    }
}
```

将导出类引用转换为基类引用的动作称为向上转型。在实现上看，导出类是基类的一个超集。在向上转型的过程中，导出类引用转换为基类引用，并且只保留基类拥有的方法。

导出类无法继承 private 函数。即使在导出类中以相同的名称声明一个函数，也不会覆盖基类中相应的 private 函数，而是生成了一个新的函数。当向上转型时，这个函数将会被丢弃。

4.2.4 继承技术的用途

相对于组合技术，继承技术不常用。只有需要从新类向基类进行向上转型，继承才是必要的。

4.3 final 关键字

final 关键字可以修饰数据、函数和类。

4.3.1 final 数据

Java 中使用 final 告知一块数据是恒定不变的，相当于 C 中的 const 关键字。需要知道的是，Java 中常量必须是基本数据类型。

一个既是 static 又是 final 的域只占据一段不能改变的存储空间。

当用 final 修饰对象引用时，这个引用将恒定不变。也就是说，引用一旦被初始化指向一个对象，就无法再把它改为指向另一个对象，而被引用的对象本身是可以被修改的。Java 没有提供使任何对象恒定不变的途径。

Java 允许生成空白 final。也就是这个域被 final 修饰但又没有赋初值。final 域在使用前必须被初始化。

在函数参数列表中将参数指明为 final，那么在函数中就无法修改参数引用所指向的对象。

4.3.2 final 函数

使用 final 函数的原因如下:

- 将函数锁定。以防任何继承类修改它的实现。
- 追求效率。当一个函数指明为 final，编译器就将该函数的所有调用都转为内嵌调用。这和 C++ 的 inline 关键字的作用一样。

类中 private 方法都隐式地指定为 final。

4.3.3 final 类

当将某个类的整体定义为 final，那么这个类就无法被继承。final 类中的域不一定是 final 的。

5 多态

多态又称为动态绑定，和 C++ 的多态类似。在讨论多态之前，先感受一下多态的特性。例子如下：

```
class Instrument
{
    public void play()
    {
        System.out.println("Instrument.play()");
    }
}

class Wind extends Instrument
{
    public void play()
    {
        System.out.println("Wind.play()");
    }
}

public class Music
{
    public static void tune(Instrument i)
    {
        i.play();
    }
    public static void main(String[] args)
    {
        Wind flute = new Wind();
        // tune接受Instrument类型
        // 将Wind转为Instrument类型
        // 输出的是: Wind.play()
        tune(flute);
    }
}
```

从这个例子可以看出一个多态的现象：虽然 tune 函数接受一个 Instrument 引用，但是它知道这个 Instrument 引用指向的是 Wind 对象。正是动态绑定实现了这项特性。

5.1 绑定

将一个函数调用和一个函数主体关联起来称为绑定。绑定有两种类型，如下：

- 前期绑定。在程序执行前就将一个函数调用和一个函数主体关联起来。
- 后期绑定，又称为动态绑定。在程序运行时根据对象的类型进行绑定。

Java 中除了 static 方法和 final 方法，其他所有方法都是后期绑定的。

5.2 多态的缺陷

5.2.1 private 函数无法动态绑定

程序不能对 private 函数进行动态绑定。这是因为 private 函数是 final 函数，而且导出类无法覆盖基类中的 private 函数。例子如下：

```
class PrivateOvrride
{
    private void f()
    {
        System.out.println("private f()");
    }

    public static void main(String[] args)
    {
        PrivateOvrride po = new Derived();
        po.f(); // 不会指向Derived类中的f()，而是指向PrivateOvrride类中的f()
    }
}

public class Derived extends PrivateOvrride
{
    public void f()
    {
        System.out.println("public f()");
    }
}
```

为了避免造成混乱的代码，导出类中的函数名不要和基类中的 private 函数名相同。

5.2.2 域无法动态绑定

和 C++ 一样，Java 中域是无法动态绑定的。也就是说，程序无法根据对象的类型选择相应的域。例子如下：

```
class Super
{
    public int field = 0;
}

Sub extends Super
{
    public int field = 1;
}

public class FieldAccess
{
    public static void main(String[] args)
    {
        Super sup = new Sub();
        System.out.println(sup.field); // 输出0
```

```
17              Sub sub = new Sub();
18              System.out.println(sub.field); // 输出1
19          }
20      }
```

为了避免造成混乱的代码，不要把基类中的域和导出类的域赋予相同的名字。

5.3 协变返回类型

协变返回类型表明，子类覆写基类方法时，返回的类型可以是基类方法返回类型的子类。

```
1   class Grain
2   {
3       public String toString()
4       {
5           return "Grain";
6       }
7   }
8
9   class Wheat extends Grain
10  {
11      public String toString()
12      {
13          return "Wheat";
14      }
15  }
16
17  class Mill
18  {
19      Grain process()
20      {
21          return new Grain();
22      }
23  }
24
25  class WheatMill extends Mill
26  {
27      Wheat process()
28      {
29          return new Wheat();
30      }
31  }
32
33  public class CovariantReturn
34  {
35      public static void main(String[] args)
36      {
37          Mill m = new Mill();
38          Grain g = m.process();
39          System.out.println(g); // 输出Grain
40          m = new WheatMill();
41          g = m.process();
```

```
42                 System.out.println(g); // 输出Wheat
43             }
44         }
```

$$\boldsymbol{H(z)=\frac{Z(z)}{p(z)}=k\frac{(z-Z(1))(z-Z(2))...(z-Z(m))}{(z-p(1))(z-p(2))...(z-p(n))}} \tag{1}$$