第一章 : 一切都是对象

-> String S : // 仅创建一个引用（无实体） -------- 未初始化

-> String S = new String("Hello world"); // 创建一个引用并指向一个实体 ------ 已初始化

-> 字段都被默认初始化为 0

-> 方法只能作为类的一部分来创建使用 -- 如果返回值不是 void , 则无论在何处返回，

编译器回强制返回一个正确类型的返回值

编码风格：

-> 类名所有单词都要大写

-> 类方法要求第一个单词小写， 其他大写

第三章 ： 操作符

算术操作符 -> 关系操作符 -> 逻辑操作符 -> 赋值

直接常量 : long(L/l) float(f/F) double(d/D) 0x(十六进制) 0(八进制)

移位操作符: <<(左移) >>(有符号右移) >>>(无符号右移)

字符号操作符：

如果表达式以一个字符串起头， 那么后续所有操作数都必须是字符串型（自动转换）

类型转换：

-> 允许八基本数据类型转成别的基本数据类型， boolean 除外

-> 类只能在所属类族间进行转化

第五章 : 初始化与清理

函数重载:

1> 方法名相同;

2> 参数不同;

3> 返回值可不相同;

说明:

-> 先寻找匹配项， 若未找到， 尝试提升

-> 字节长度小于 int 会优先尝试提升到 int

整形会优先提升到 float 型

成员初始化：

-> 字段会首先默认初始化(如未指定， 则初始化为0)

而后再使用构造器构造

-> 局部变量不会被自动初始化

-> 静态数据 (static):

1> 不能应用与局部变量

2> 如果为被显示初始化， 则默认为 0

3> 当首次生成一个类对象或首次访问属于那个类的静态成员或方法时，

所有的静态成员被初始化

说明：

1> 初始顺序是先静态后非静态成员

2> 构造器是 static 方法

枚举 :

// 枚举本质是一个类

public enum EnumName{NOT, MILD, ..}

方法:

toString() // 显示枚举名

ordinal() // 表示出其顺序值

values() // 产生这些常量值构成的数组

第六章：访问权限控制

-> 访问权限修饰词：

private : 只能在本类中访问

包访问权限: 只能在本包中访问

protected: 处本包外， 还可在外包继承类中访问

public : 对外完全开放访问权限

-> 类的访问权限

类不能指定为private 或 protected;

默认为包访问权限， 即只能在本包中访问

public 外包可使用

第七章 复用类

→ 继承语法：

除非已明确的指出要从其他类中继承， 否则就隐式的从 Object 类中继承

→ 初始化基类：

1> 构造顺序从基类到派生类

2> 如果为显式调用基类的构造器， 默认调用调用基类的无参构造器

3> 如果要显式的调用 super(), 则应该在构造器的开始处调用。

→ 确保正确被清理：

→ 除了内存意外， 不能以来垃圾回收器去做任何事， 如果需要进行清理， 必须手动编写， 不要使用 finalize();

→ 名字屏蔽：

→ 如果基类特征标签不同的方法不会屏蔽基类的方法， 只能构成重载， 如果要避免以上行为， 可以使用 @Override 注释

→ final 关键字

1> 用于字段（类似于 C 中的 const）：

a> 对于一般的成员类型，成员值无法改变 ;

b> 对于引用， 则引用无法在指向其他的对象;

说明 ：

必须在构造器或者定义处为其赋值， 但不能同时使用.(二次定义问题)

2> 用于方法： 方法不能再被覆写， (private → private final)

3> 用于类： 类无法再被继承， 同时所有的方法默认添加 final ；

第八章 ：多态

→ 方法调用绑定:

java 中除了 static 方法和 final 方法外， 其他方法都采用后期绑定， 静态方法允许覆写， 但不具备多态 性

→ 静态方法是与类， 而不是与类对象关联;

→ 构造器和多态

构造器的调用顺序：

→ 在其他任何事物发生之前， 将分配给存储空间初始化成二进制的零

1> 调用基类构造器， 这个步骤会不断反复递归下去， 首先是构造这种层次结构的根， 然后是一层类。。

2> 顺序调用成员初始化方法。

3> 调用基类构造器主体

→ 清理顺序与构造顺序相反

第九章 接口

→ 抽象类和抽象方法 （abstract）

如果从一个抽象类继承， 并想创建该新类的对象， 那么就必须为基类中的所有抽象方法提供方法定义。如果不 这样左， 那么派生类便必须也声明为抽象类

→ 接口( interface ) – implements

方法: public abstract

字段： public static final

功能 : 为了能够向上转型为多个基类型， 防止客户端程序员创建该接口的对象

→ 工厂方法模型

第十章 内部类

说明：

1> 普通的内部类对象隐式的保存了一个引用， 指向创建他的外围设备对象.

2> 内部类不能有 static 字段跟 static 方法

→ 创建内部类：

→ 链接到外部类：

1> 内部类能够访问其外围对象的所有成员， 而不需要任何条件。外部类也可直接访问内部类对象

2> 当莫个外围类的对象创建一个内部类对象时， 此内部类对象必须会秘密的捕获到其相关联的外部类对 象

→ 使用 .new 与 .this

对象名+ .new （用于内部类的内部方法调用） ： 用于生成外部类的引用

类名 + .this（用于外部调用）

→ 将创建的内部类对象关联到某个外部类对象上。

→ 在创建内部类对象之前必须先创建外部类对象， 因为内部类对象一定要链接一个外部类对象

→ 内部类的向上转型

使用继承或实现接口

→ 在方法和作用域的内部类：

使用原因：

1> 使用类可以实现了某种类型的接口， 于是可以创建并返回对其引用

2> 内部类用来辅助解决问题， 又不希望其在外部可用

使用的方式：

1> 定义在方法与作用域中的类。

2> 匿名类：

→ 允许字段初始化。

→可以通过实例初始化实现构造

限制：

1> 匿名类既可以扩展类， 也可以实现接口， 但只能择其一， 而且最多只能实现一个接口

2> 如果定义一个匿名类， 并在该类中使用外部对象， 则应该声明为 final 类型

→ 嵌套类(static)

→ 与外围之类之间相互独立

→

1> 要创建嵌套的对象， 并不需要外围类对象

2> 不能从嵌套类中访问外部非静态类对象

→ 嵌套类可以做为接口的一部分， 放在接口中的类默认为 public static

第十一章 ： 持有对象

→ 泛型和类型安全的函数:

→ ArrayList:

→ ArrayList<>

→ 基本概念:

两种容器的类型

1> Collection : List, Set, Queue;

2> Map:

→ List ( List<T> ):

ArrayList: 用于随机访问， 但在中间插入和删除元素的代价较大

LinkedList: 用于在List中间进行插入和删除操作， 但是随机访问代价较大

→ 迭代器:

→ Iterator : 使用 iterator() 返回一个对象， 该队项准备返回队列的第一个元素;

→ next(); 返回序列中的下一个元素

→ hasNext(); 检测是否还有元素

→ remove(); 将新近元素删除

→ ListIterator: // 只能访问 List 的各种类型

支持双向遍历

→ Stack:

java.util 中没有任何公共的stack 接口, 可以通过封装 LinkedList 来实现自己的类型

→ Forech 与 迭代器

Iteratable 的接口的引入， 该接口能够产生Iterator的iterator方法， 并且Iterable 接口被 foreach 用来在序列中移动

说明：

→ 传递一个 Iterable 的接口调用， 然后调用该接口的iterator 的方法， 该方法返回一个 Iterator的迭代器

第12章： 通过异常处理错误

→ 基本异常：

→ throw : 抛出一个异常对象;

→ 异常对象可以接收一个 String 类型的参数;

→ 捕获异常： try … catch块

→ 每一个 catch 子句接收且仅接收一个特殊类型的参数方法。

→ 异常处理程序必须紧跟 try 之后。

→ 创建自定义异常：

→ printStackTrace(); // 默认打印到标准错误流;

→ 异常与记录日志：

→ getMessage(); // 返回出错的详情调用信息， 被 printStackTrace()调用

→ 异常说明：

→ throws 声明抛出的异常类型， 实际可并不抛出;

→ 捕获所有的异常 :

→ 使用 Exception ， 最好把它的 catch 放在末尾处;

→ 重新抛出异常

→ 如果只是把当前异常对象重新抛出， 那么printStackTrace 返回的是原来异常抛出点的调用信息， 而 并非重新抛出点的调用信息

→ 使用 fillInStackTrace();

→ 抛出另一种异常;

→ RuntimeException:

→ 由java虚拟机自动抛出， 所以不必在异常说明中将他们列出来

→ 会自动被捕获， 所以可不必手动写 try … catch 块

→ 如果是自动被捕获， 默认在结束位置调用 printStackTrace();

→ finally:

作用： 把除内存之外的资源恢复到初始状态

→ 当涉及 break 和 continue 语句， return 语句时， finally 子句还是会被执行

→ 异常丢失:

1> 在 finally 中返回新的异常

2> 在 finally 中调用 return

→ 限制 （覆写）

只能抛出那些在基类方法中列出的那些异常

**13- Class 类**

-> 没有公共的构造方法， 所以不能使用 Class cls1 = new Class ();

构造方法方法(获取子节码) :

→ className.class, 例如: System.class;

→ 对象名.getClass(), 例如: new String().getClass();

→ Class.forName(“className”); 例如: Class.forName(“java.util.Date”);

→ 使用forName 调用的类名要用全名

-> 构造过程 :

-> 首先检查这个类是否被加载， 如果未被加载， 则尝试加载 .class 文件

-> 如果已经被加载， 则直接从内存中直接引用

-> 对同一个类生成不同的 Class 对象， 则只引用同一个内存对象

→ 在程序中出现的所有的变量， 都有与之对应的 Class 对象

反射：

Constructor: // 代表某个类中的一个构造方法

→ 通过 Class 的 getConstructor(Object... args); // 获取构造器

→ 通过 Constructor 的 newInstance(); // 来使用获取到的构造器

→ // 需要传递同样类型的对象

Field : // 代表类的字段

// 通过变量名来获取类变量

→ getField(); // 获取其允许访问的成员字段

→ getDeclaredField(); // 可以获取 private 的成员

→ setAccessible(); // 设置 private 字段的访问权限

→ get(); // 获取此字段指定对象的字段

-> set(); // 设置此变量的值

Method: // 代表类的方法

**14- 注解**

@Deprecated // 注明过时

@Override // 防止重载

@SuppressWarning / / 抑制警告

说明：

→ 注解相当于一种标记， 在程序中加了注解就等于为程序打上了某种标记， 没加， 则等于没有某种标记， 以后， javac 编译器， 开发工具和其他程序可以用反射来了解你的各种元素上有何种标记， 就去干 相应的事。

→ 标记可以加在包， 类， 子段， 方法， 方法的参数以及局部变量上

→ 为注解增加属性:

@Rertention 用于标注自定义注解的生命周期

@Target 用于指明注解的使用领域

**15-泛型**

→ 说明：

1> 泛型与原始类型的互兼容性

2> 参数化的类型不考虑类型参数的兼容性

3> 在创建数组实例时， 数组的元素不能使用参数化的类型

-> 注意：

1> 在对泛型类型进行参数化时， 类型参数的实例必须是引用类型， 不能是基本类型

-> 泛型的主要目的就ishi指定容器要持有什么类型的对象， 而且由编译器来确保类型的正确性

泛型接口：

-> 一般用于生成器(专门负责创建对象的类)， 这是工厂方法设计模式的一种应用， 一个生成器只有一个方法， 该方 法用于产生对象

泛型方法：

注意：

→ 当一个变量被声明为泛型时， 只能被实例变量和方法调用（还有内嵌类型）， 而不能被静态类型和静 态方法调用， 因为静态成员是被所有参数化的类所共享的， 所以静态成员不应该有类级别的类型参数；

-> 自动类型推断只对赋值操作有效

→ 如果在函数调用内要使用泛型方法 要显式类型说明

→ 显式类型说明：

点操作符与方法之间添加实例类型

擦除问题：

→ 泛型类型只有在静态类型检查期间才会出现， 在此之后程序中的所有泛型类型都将被擦除， 替换为他们的非泛型 上界

→ 普通的类型变量在未指明边界的情况下都会被擦除为 Object 。

→ 擦除是非泛化的代码到泛化的代码的迁移

擦除的补偿：

→ 关于类型信息丢失问题：

→ instanceof 已经无法调用 ， 使用动态的 isInstance()来解决

→ new 表达式无法使用，

1> 通过传递 class 对象调用 newInstance 方法， 或者Constrict 的方法

2> 通过使用基于接口的泛型构造器

3> 通过使用模版方法来创建类型实例

泛型数组：

1> 不能创建一个泛型实例的数组(ArrayList<Integer>[] args)， 一般使用容器来解决该问题

2> 可以在泛型中使用数组 （private T[] array;）

通配符：

→ 泛型不会因存在继承关系而同意赋值操作：

→ 向上转型： <? extends MyClass>

→ 逆转： <? super MyClass>

→ 无边界通配符 <?>

**数组**

→ 数组是一种效率最高的存储和随机访问对象引用序列的方式；

说明：

数组标识符其实只是一个引用， 指向在堆中创建的一个真实对象

对象数组保存的是引用

基本类型数组保存的是实体

初始化：

1> 直接赋值传递

2> 通过花括号来分配并初始化成员

3> 通过new 来分配并默认初始化

4> 通过 new + 花括号来分配

固定成员：

length :

可变参数：

例如： int... args; for(int once : args)

Arrays 的实际功能

**容器**

→ Iterable 用法：用于 foreach 语句中, 一般用于容器中

特点：

→ 首先容器必须要实现 Itarable 接口， 实现其 iterator 方法

→ Collection 的功能和方法：

可选操作：

1> UnSupportedOperationException 必须是一种罕见的事件， 即对大多数来说， 所有的操作都应该得到支持， 只有特例才会有未支持的操作；

2> 如果一种操作为获得支持， 那么在实现接口时可能会导致 UnspportedOperationException 异常

3> 未获得支持的操作只在运行时才能监测到 → 动态类型检查

→ List 功能方法：

→ LinkedList:

→ ArrayList:

→ Set 和 存取顺序：

→ HashSet → LinkedHashSet

→ TreeSet(SortSet)

1> 存入 Set 中的每一个元素必须是唯一的， Set 不保存重复元素 → 使用 equals() 方法来确定唯一性

2> HashSet 实现了快速查找 → 容器中的元素必须定义有 hashCode();

3> LinkedHashSet 使用链表维护其次序

4> TreeSet, 底层为树结构， 可以提取为 有序队列 → 容器中的元素必须实现 Comparable 接口

**Java I/O 系统**

→ File 类：

→ 输入和输出：

1> ByteArrayInputStream // 将内存的缓冲区当作 InputStream使用 ----------------- CharArrayReader

ByteArrayOutputStream // 将数据写入内存 ----------------- CharArrayWriter

2> StringBufferInputStream // 将String 转成 InputStream

3> FileInputStream // 用于从文件中读取信息 ------------------ FileReader

FileOutputStream // 将信息写入文件 ------------------ FileWriter

4> PipedInputStream // 管道化 ------------------ PipedReader

PipedOutputSteam // 管道化 ------------------ PipedWriter

5> FilterInputStream // 装饰器 ------------------ FIlterReader

→ DataInputSteam

→ BufferedInputStream ------------------- BufferedReader

→ LineNumberInputStream

→ PushbackInputStream

FilterOutputStream // 装饰器 ------------------- FilterWriter

→ DataOutputStream

→ BufferedOutputStream ------------------- BufferedWriter

→ PrintStream

6> RandomAccessFile // 随机读写文件的类

-> 标准IO

1> System.in // 标准输入 （需要手动封装） → InputStreamReader

2> System.out(PrintStream) //标准输出 → OutputStreamWriter

3> System.err // 标准错误

→ 标准 IO的重定向

1> setIn(InputStream)

2> setOut(OutputStream)

3> setErr(PrintStream)

→ 进程控制：

→ ProcessBulider =》 Process

→ getInputStream();

→ getOutputStream();

→ getErrorStream();

→ 新I/O类

ByteBuffer → 直接与通道相连的缓冲器

文件读写: FileChannel ← → ByteBuffer

**线程**

创建新的线程：

1 > Runnable

-> 定义线程类实现 Runnable 接口

→ Thread myThread = new Thread(target); // target 为Runnable 接口类型

→ Runnable 只有一个方法：

public void run(): //线程体

→ 使用 Runnable 接口的类可以为多个线程提供共享数据

→ 在实现 Runnable 接口的类的run 方法定义中可以使用 Thread 静态方法

→ public static Thread currentThread(); // 获得当前的调用线程

2> Thread

→ 可以定义一个 Thread 子类并重写其 run 方法如：

class MyThread extends Thread{

public void run(){..............};

}

然后生成该类对象并调用 start 方法

3> Executor 类

1.**newCachedThreadPool**()

-缓存型池子，先查看池中有没有以前建立的线程，如果有，就reuse.如果没有，就建一个新的线程加入池中

-缓存型池子通常用于执行一些生存期很短的异步型任务， 因此在一些面向连接的daemon型SERVER中用 得不多。

-能reuse的线程，必须是timeout IDLE内的池中线程，缺省timeout是60s,超过这个IDLE时长，线程实例 将被终止及移出池。

注意，放入CachedThreadPool的线程不必担心其结束，超过TIMEOUT不活动，其会自动被终止

2. **newFixedThreadPool**

-newFixedThreadPool与cacheThreadPool差不多，也是能reuse就用，但不能随时建新的 线程

-其独特之处:任意时间点，最多只能有固定数目的活动线程存在，此时如果有新的线程要建立，只能放在另 外的队列中等待，直到当前的线程中某个线程终止直接被移出池子

-和cacheThreadPool不同，FixedThreadPool没有IDLE机制（可能也有，但既然文档没提，肯定非常长，类 似依赖上层的TCP或UDP IDLE机制之类的）, 所以FixedThreadPool多数针对一些很稳定很固定的正 规并发线程，多用于服务器

-从方法的源代码看，cache池和fixed 池调用的是同一个底层池，只不过参数不同:

fixed池线程数固定，并且是0秒IDLE（无IDLE）

cache池线程数支持0-Integer.MAX\_VALUE(显然完全没考虑主机的资源承受能力），60秒IDLE

3.**ScheduledThreadPool** -调度型线程池

-这个池子里的线程可以按schedule依次delay执行，或周期执行

4.**SingleThreadExecutor** -单例线程，任意时间池中只能有一个线程

-用的是和cache池和fixed池相同的底层池，但线程数目是1-1,0秒IDLE（无IDLE）

线程控制的方法：

1> isAlive // 判断线程是否还存活

2> getPriority() // 获得线程的优先级

3> setPriority() // 设置线程的优先级数值

4> sleep() // 将当前线程睡眠指定毫秒数（static）

5> join() // 合并线程

6> yield() // 让出某个线程

7> wait() // 当前线程进入对象 的 wait pool

8> notify()

9> notifyAll(); //唤醒 wait 对象

同步机制：

→ synchronized // 互斥锁 关键字

1> Java语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。

一、当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程 必须等待当前线程执行完这个代码块以后才能执行该代码块。

二、然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同 步代码块。

三、尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步 代码块的访问将被阻塞。

四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个 object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。

五、以上规则对其它对象锁同样适用.

2> synchronized 关键字，它包括两种用法：synchronized 方法和 synchronized 块。

a>synchronized 方法：通过在方法声明中加入 synchronized关键字来声明 synchronized 方法。如：

public synchronized void accessVal(int newVal);

synchronized 方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须 获得调用该方法的类实例的锁方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该 方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。

b> synchronized 块：通过 synchronized关键字来声明synchronized 块。语法如下：

synchronized(syncObject) {

//允许访问控制的代码

}

synchronized 块是这样一个代码块，其中的代码必须获得对象 syncObject （如前所述，可以是类实例或类）的锁 方能执行，具体机制同前所述。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。

→ 使用显示的 Lock 对象和 Condition 对象

说明 显示的 使用 Lock 对象时， 最好把 Lock.unlcok 放在 finally 语句中

→ 中断：

1. interrupt 方法调用: 注意处理异常

2. 使用cancel（） :

→ 不能中断任何 synchronzed 锁, 或者试图执行 IO 操作的进程