## 一 Java 局部内部类使用局部final变量

1. 内部类是外部类的一个成员，就像外部类的成员方法一样，所以内部类有权限访问外部类的所有成员，包括private的。
2. 内部类不能访问外部类方法中的局部变量，除非变量是final的(一般发生在方法中定义的内部类)。这是因为局部变量的生命周期原因。
3. 所谓“局部内部类”就是在对象的方法成员内部定义的类。而方法中的类，访问同一个方法中的局部变量，却必须要加上一个final。

原因是编译程序实现上的困难：

* 内部类对象的生命周期会超过局部变量的生命期。局部变量的生命期：当该方法被调用时，该方法中的局部变量在栈中被创建，当方法调用结束时，退栈，这些局部变量全部死亡。而内部类对象生命期，与其它类一样，当创建一个局部内部类对象后，只有当没有其它人再引用它时，它才能死亡。所以完全可能一个方法已调用结束（局部变量已死亡），但该局部类的对象仍然活着。即：局部类的对象生命期会超过局部变量。
* 局部内部类的对象访问同一个方法中的局部变量，那么这就要求只要局部内部类对象还活着，那么栈中的那些它要访问的局部变量就不能“死亡”（否则：它都死了，还访问个什么呢？）。这就是说：**局部变量的生命期至少等于或大于局部内部类对象的生命期。**
* 解决方法：局部内部类的对象可以访问同一个方法中被定义为final的局部变量。定义为final后，编译程序的实现方法：将所有的局部内部类对象要访问的final型局部变量，都**拷贝**成为该内部类对象中的一个数据成员。这样，即使栈中局部变量（含final）已死亡，但由于它是final,其值永不变，因而局部内部类对象在变量死亡后，照样可以访问final型局部变量
* 归纳总结：局部内部类对象中包含有要访问的final型局部变量的一个**拷贝**，成为它的数据成员。因此，正是在这个意义上，final型局部变量的生命期，超过其方法的一次调用。**严格来说，方法调用结束，所有的局部变量（含final）全死亡了。但：局部内部类对象中有final型局部变量的拷贝。**

## 二 java 24 \* 60 \* 60 \* 1000（一天）

Date d=new Date();

SimpleDateFormat df=new SimpleDateFormat("yyyy-MM-dd");

System.out.println("today:"+df.format(d));

System.out.println("the day **befor yestoday**:" + df.format(new Date(**d.getTime() - (long)2 \* 24 \* 60 \* 60 \* 1000)**));

System.out.println("the day **after tomorrow**:" + df.format(new Date(**d.getTime() + (long)3 \* 24 \* 60 \* 60 \* 1000)**));

## 三 Java中子类和父类的初始化顺序-静态优先，父类优先

类的初始化顺序是：

1、初始化父类中的静态成员变量和静态代码块。

2、初始化子类中的静态成员变量和静态代码块。

3、初始化父类中的普通成员变量和代码块，在执行父类中的构造方法。

4、初始化子类中的普通成员变量和代码块，在执行子类中的构造方法。

## 四 Java transient关键字

java语言的关键字，变量修饰符，如果用transient声明一个实例变量，当对象存储时，它的值不需要维持。换句话来说就是，用transient关键字标记的成员变量不参与序列化过程。

## 五 Java >> n & << n

>> n （右移 除2的n次方）

<< n （左移 乘2的n次方）

## 六 Java Collection接口的toArry方法注意

1. public Object[] toArray()方法：方法中会重新构造一个Object[]数组，然后将集合中每个元素赋给数组，由于new的是Object数组，所以无法向上转型回之前的数组。
2. public <T> T[] toArray(T[] a)方法：返回传入的数组a指定的类型，a的length小于集合size时会利用反射重新构建一个指定类型数组，并把集合中所有元素赋给数组。

## 七 编程习惯

### 异常处理

<http://www.importnew.com/26858.html>

不要随便加null判断（并不“健壮”，反而掩盖错误）：如果写了，就必须测试为空和不为空的两种情况

不捕获异常，将异常交给调用者处理：比起从日志中寻找错误，应在编码阶段处理，尽早让错误抛出。

### 接口定义

<https://zhuanlan.zhihu.com/p/28708259>

### Controller规范

## 八 Java 8

<http://www.importnew.com/16436.html>

<http://www.importnew.com/11908.html>

1. Lambda表达式（闭包）；
2. 函数式编程支持： java.util.function.\* 下的众多接口；
3. 流API；
4. 接口的默认方法和静态方法；
5. 新的Date和Time API。

### 1 Optional 接口

<http://www.importnew.com/6675.html>

对对象的null值检测进行包装，如果值存在则isPresent返回true，否则false。

Optional.of：创建一个Optional，不能为null，赋值抛出NullPointerException。

Optional.ofNullable：可以创建一个null的Optional。

isPresent：值存在返回true，否则false。

ifPresent：值存在就调用Consumer对象的accept方法，否则无任何动作。

get：返回值，空Optional抛NoSuchElementException异常。

orElse：值存在返回，否则返回指定值。

orElseGet：值存在返回，否则调用Supplier的get返回。

orElseThrow：存在返回，否则抛出指定异常。

map、flatMap、filter

### 2 Lambda

Lambda表达式允许将函数作为一个方法参数传递，或许将代码看成数据。

用 () -> {} 代替整个匿名类。

new Thread( **() -> {**int i = 9 ; System.out.println(“somting”+i) ; }).start();

() 中参数可省略类型，没有参数时 () 仍然要保留。

{} 只有一条语句时 {} 可以省略，语句尾的 ; 也可以省略。

Arrays.asList(“a”, “b”, “c”, “d”).forEach(va -> System.out.println(va));

**FunctionalInterface：函数式接口，如果一个接口只有一个方法，那么编译器会将该接口视为函数式接口，如果在接口上加注解FunctionalInterface注解，那么接口就会被强制要求符合函数式接口的规范（只有一个抽象方法，默认，静态方法不包括在内）。**

### 3 Predicate接口

适用于过滤。

该接口有一个test方法，返回boolean，表示检查结果。提供了类似于逻辑操作符AND和OR的方法and()、or()、xor()。可以将多个条件进行组合。

Predicate<String> testStart = (str) -> str.startsWith("a");  
Predicate<String> testContain = (str) -> str.contains("b");  
boolean result = testStart.and(testContain).test("acvvb"); // true

### 4 Map的Reduce

map方法（Function接口）遍历并将每一个数乘以2，reduce（BiFunction接口）方法将结果汇总于sum。

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);  
Optional<Integer> reduce = list.stream().map(va -> va >> 1).reduce((sum, cur) -> sum += cur);  
*o*.accept(reduce.orElse(-1));

### 5 Function<T,R>接口

用于产生对象，函数式接口，接受一个参数，构造并返回目标对象。

R apply(T t);

### 6 BiFunction<T,R,U>接口

用于产生对象，接受两个参数，生成一个结果。

R apply(T t,U u);

### 7过滤集合元素

Filter方法（Predicate接口）遍历元素并检查，返回true保留，collect方法（Collector接口）将结果组合为新的集合。

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);  
List<Integer> collect = list.stream().filter(va -> va % 2 == 0).collect(Collectors.*toList*());  
*o*.accept(collect);

### 8 Supplier接口

用来产生对象

T get();

### 9 计算集合元素的最大，最小，总和以及平均值

IntStream、LongStream和DoubleStream中有个summaryStatics方法，可以返回IntSummaryStatics、LongSummaryStatics或DoubleSummaryStatics，描述流中的各种摘要数据。

Stream的mapToInt/Long/Double方法可将指定对象作为参数，产生对应的基本类型，mapXXX中接受的接口为：ToInt/Long/DoubleFunction，方法为(以int为例)：int applyToInt(T value);

List<Integer> list = Arrays.*asList*(1, 2, 3, 4, 5, 6);  
IntSummaryStatistics statistics = list.stream().mapToInt(i -> i).summaryStatistics();  
statistics.getMax();  
statistics.getMin();  
statistics.getAverage();  
statistics.getSum();  
statistics.getCount();

### 10 Lambda VS 匿名内部类

this关键字：匿名类的this指向匿名类，而Lambda的this指向外部类。

编译方式：Lambda被编译为外部类的私有成员方法，而匿名类被编译为单独的类。

同：两种方式在引用外部的局部变量时局部变量要被final修饰。

### 11 方法引用

List<Car> cars = new ArrayList<>();

1. 方法构造器（无参可访问的构造器）引用：Class::new ，可以便捷的创建一个对象，通常配合Supplier接口使用。
2. 引用静态方法（一参可访问）：Class::static\_method，通常配合Consumer接口使用。
3. 引用特定类的方法（无参可访问）：Class::method
4. 引用特定对象的方法：instance::method

### 12 新的Date/Time API

在java.time.\*包下：

新的API涵盖了所有处理日期、时间、时区、过程与时钟的操作。

1. Clock类：

UTC（Universal Time Coordinated）为世界协调时间，与格林尼治时间一样，与英国伦敦时间也一样。北京是东八区，领先于UTC8个小时。

Clock clock = Clock.*systemUTC*();  
 Instant instant = clock.instant();  
 Instant plus = instant.plus(8, ChronoUnit.*HOURS*);// 加 8 小时  
  
 Instant minus = instant  
// .plus(24 \* 60 \* 60 \* 1000, ChronoUnit.MILLIS) // 加一天  
 .minus(24 \* 60 \* 60 \* 1000, ChronoUnit.*MILLIS*);// 减一天  
 *o*.accept(instant); // UTC 时间：2017-10-24T07:11:12.576Z  
 *o*.accept(plus);// 北京时间：2017-10-24T15:11:12.576Z  
 *o*.accept(minus);// UTC 时间减一天：2017-10-23T07:11:12.576Z

1. LocalDate、LocalTime和LocalDateTime

三者只持有ISO-8601格式具有时区信息的日期与时间。前两者都可以从Clock中获得。

Clock clock = Clock.*systemUTC*();  
LocalDate date = LocalDate.*now*();  
LocalTime time = LocalTime.*now*(clock);  
LocalDateTime dateTime = LocalDateTime.*now*();  
*o*.accept(date); //2017-10-24  
*o*.accept(time); // 07:22:07.781 系统时间：15:22:37.050  
*o*.accept(time.getNano()); // 319000000  
*o*.accept(dateTime); // 2017-10-24T15:23:54.109

1. Duration，在秒与纳秒级别上的一段时间。简化了计算两个时间差的不同。

Duration duration = Duration.*between*(  
 LocalDateTime.*of*(2014, Month.*JULY*, 13, 21, 0, 0), // 2014-07-13 21:00:00  
 LocalDateTime.*of*(2014, Month.*JULY*, 14, 22, 13, 16)); // 2014-07-14 22:13:16  
*o*.accept(duration.toDays()); // 1  
*o*.accept(duration.toHours()); // 25  
*o*.accept(duration.toMinutes()); // 1513  
*o*.accept(duration.toNanos()); // 90796000000000

## 九 java值传递

Java使用对象的引用进行计算，但是java在向方法传递对象时传递的不是引用，是值。

常见的swap函数：

public void swap(int a,int b) { // 错误的交换方法

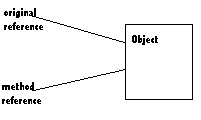
int temp = a;

a = b;

b = temp;

}

但swap方法返回时，a，b并没有交换。Java通过传值来传递引用：这意味着，**传向函数的引用实际上是原始引用的副本**。进行交换操作时，交换的是副本引用，真正的引用并未交换。（我们应该交换原始的引用，而不是它们的副本）。



当函数被调用后，一个对象至少存在两个引用。

## 十jvm 的内存中可以把一个类加载多次么

<https://zhidao.baidu.com/question/1899587421534177660.html>