【软件构造】期末知识点

【考点 Equals】

- ==是引用等价性； 而equals()是对象等价性。
  - == 比较的是索引。更准确的说，它测试的是指向相等（referential equality）。如果两个索引指向同一块存储区域，那它们就是==的。对于我们之前提到过的快照图来说，==就意味着它们的箭头指向同一个对象。
  - equals()操作比较的是对象的内容，换句话说，它测试的是对象值相等（object equality）。e在每一个ADT中，quals操作必须合理定义。
- 基本数据类型，也称原始数据类型。byte,short,char,int,long,float,double,boolean
  - 他们之间的比较，应用双等号（==）,比较的是他们的值。
- 复合数据类型(类)
  - 当他们用（==）进行比较的时候，比较的是他们在内存中的存放地址，所以，除非是同一个new出来的对象，他们的比较后的结果为true，否则比较后结果为false。
  - JAVA当中所有的类都是继承于Object这个基类的，在Object中的基类中定义了一个equals的方法，这个方法的初始行为是比较对象的内存地址，但在一些类库当中这个方法被覆盖掉了，如String,Integer,Date在这些类当中equals有其自身的实现，而不再是比较类在堆内存中的存放地址了。
  - 对于复合数据类型之间进行equals比较，在没有覆写equals方法的情况下，他们之间的比较还是基于他们在内存中的存放位置的地址值的，因为Object的equals方法也是用双等号（==）进行比较的，所以比较后的结果跟双等号（==）的结果相同。
- HahCode：Java中的hashCode方法就是根据一定的规则将与对象相关的信息（比如对象的存储地址，对象的字段等）映射成一个数值，这个数值称作为散列值。

当我们当向集合中插入对象时，就可以使用hashcode，先调用这个对象的hashCode方法，得到对应的hashcode值，实际上在HashMap的具体实现中会用一个table保存已经存进去的对象的hashcode值，如果table中没有该hashcode值，它就可以直接存进去，不用再进行任何比较了；如果存在该hashcode值， 就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址，所以这里存在一个冲突解决的问题，这样一来实际调用equals方法的次数就大大降低了。

- · Java中的equals与hashCode方法(判断插入容器的内容是否重复)
- · String和equals()、hashCode()
- · java中对象的equals和hashcode覆盖原则
- · HashMap中的equals和hashCode
- · java的equals、hashcode和Clone方法
- · Java中的equals和hashCode方法详解
- · 学习笔记-JAVA-考点10-什么情况下需要重写equals和hashcode()两个方法？

【考点 函数规约】 requires与effects

声明式规约更有价值； 内部实现的细节不在规约里呈现，而放在代码实现体内部注释里呈现，例：

```
static String join(String delimiter, String[] elements)
```

```
effects: returns concatenation of elements in order, with delimiter inserted between each pair of
adjacent elements // Declarative specs
```

更强的规约包括更轻松的前置条件和更严格的后置条件；

方法前的注释也是一种规约，但需人工判定其是否满足。

- 参数由@param 描述
- 子句和结果用 @return 和 @ throws子句 描述
- 尽可能的将前置条件放在 @param 中
- 尽可能的将后置条件放在 @return 和 @throws 中

```
/**
 * Find a value in an array.
 * @param arr array to search, requires that val occurs exactly once
 *            in arr
 * @param val value to search for
 * @return index i such that arr[i] = val
 */
static int find(int[] arr, int val)
```

【考点 **ADT**的四种类型】

- Creators（构造器）：
  - 创建某个类型的新对象，一个创建者可能会接受一个对象作为参数，但是这个对象的类型不能是它创建对象对应的类型。可能实现为构造函数或静态函数。（通常称为工厂方法）
  - t* -> T
  - 栗子：Integer.valueOf(Object obj)：object → integer
- Producers（生产器）：
  - 通过接受同类型的对象创建新的对象。
  - T+ , t* -> T
  - 栗子：String.concat( ) ： String x String → String
- Observers（观察器）：
  - 获取抽象类型的对象然后返回一个不同类型的对象/值。
  - T+ , t* -> t
  - 栗子：List.size( ) ： List → int
- Mutators（变值器）：
  - 改变对象属性的方法，
  - 变值器通常返回void，若为void，则必然意味着它改变了对象的某些内部状态；当然，也可能返回非空类型
  - T+ , t* -> t || T || void
  - 栗子：List.add( )： List x int → List

【考点 **ADT**的 **AF**与 **RI**】

在研究抽象类型的时候，先思考一下两个值域之间的关系：

- 表示域（rep values）里面包含的是值具体的实现实体。一般情况下ADT的表示比较简单，有些时候需要复杂表示。
- 抽象域（AF）里面包含的则是类型设计时支持使用的值。这些值是由表示域**“抽象/想象”**出来的，也是使用者关注的。

R->A的映射特点：

- 每一个抽象值都是由表示值映射而来，即满射：每个抽象值被映射到一些rep值
- 一些抽象值是被多个表示值映射而来的，即未必单射：一些抽象值被映射到多个rep值
- 不是所有的表示值都能映射到抽象域中，即未必双射：并非所有的rep值都被映射。

在描述抽象函数和表示不变量的时候，注意要清晰明确：

- 对于RI（表示不变量），仅仅宽泛的说什么区域是合法的并不够，你还应该说明是什么使得它合法/不合法。
- 对于AF（抽象函数）来说，仅仅宽泛的说抽象域表示了什么并不够。抽象函数的作用是规定合法的表示值会如何被解释到抽象域。作为一个函数，我们应该清晰的知道从一个输入到一个输入是怎么对应的。

```
ublic class RatNum {

  private final int numer;
  private final int denom;

  // Rep invariant:
  //   denom > 0
  //   numer/denom is in reduced form

  // Abstraction Function:
  //   represents the rational number numer / denom

  /** Make a new Ratnum == n.
   *  @param n value */
  public RatNum(int n) {
      numer = n;
      denom = 1;
      checkRep();
  }
```

【考点：黑盒、白盒框架】

- 框架（Framework）是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法;另一种定义认为，框架是可被应用开发者定制的应用骨架。前者是从应用方面而后者是从目的方面给出的定义。
- 为了增加代码的复用性，可以使用委派和继承机制。同时，在使用这两种机制增加代码复用的过程中，我们也相应地在不同的类之间增加了关系（委派或继承关系）。而对于一个项目而言，各个不同类之间的依赖关系就可以看做为一个框架。一个大规模的项目可能由许多不同的框架组合而成。

【白盒框架】

- 白盒框架是基于面向对象的继承机制。之所以说是白盒框架，是因为在这种框架中，父类的方法对子类而言是可见的。子类可以通过继承或重写父类的方法来实现更具体的方法。
- 虽然层次结构比较清晰，但是这种方式也有其局限性，父类中的方法子类一定拥有，要么继承，要么重写，不可能存在子类中不存在的方法而在父类中存在。
- 软件构造课程中有关白盒框架的例子：

```
public abstract class PrintOnScreen {
    public void print() {
        JFrame frame = new JFrame();
        JOptionPane.showMessageDialog(frame, textToShow());
        frame.dispose();
    }
    protected abstract String textToShow();
}
```

```
public class MyApplication extends PrintOnScreen {
@Override protected String textToShow() {
        return "printing this text on " + "screen using PrintOnScreen " + "white Box Framework";
    }
}
```

【黑盒框架】

- 黑盒框架时基于委派的组合方式，是不同对象之间的组合。之所以是黑盒，是因为不用去管对象中的方法是如何实现的，只需关心对象上拥有的方法。
- 这种方式较白盒框架更为灵活，因为可以在运行时动态地传入不同对象，实现不同对象间的动态组合；而继承机制在静态编译时就已经确定好。
- 黑盒框架与白盒框架之间可以相互转换，具体例子可以看一下，软件构造课程中有关黑盒框架的例子，更改上面的白盒框架为黑盒框架：

```
public interface TextToShow {
    String text();
}
```

```
public class MyTextToShow implements TextToShow {
    @Override
    public String text() {
        return "Printing";
    }
}
```

```
public final class PrintOnScreen {
    TextToShow textToShow;
    public PrintOnScreen(TextToShow tx) {
        this.textToShow = tx;
    }
    public void print() {
        JFrame frame = new JFrame();
        JOptionPane.showMessageDialog(frame, textToShow.text());
        frame.dispose();
    }
}
```

【两者对比】

- 白盒框架利用**subclassing**：
  - 允许扩展每一个非私有方法
  - 需要理解父类的实现
  - 一次只进行一次扩展
  - 通常被认为是开发者框架
- 黑盒框架使用委派中的组合**composition**：
  - 允许在接口中对public方法扩展
  - 只需要理解接口
  - 通常提供更多的模块
  - 通常被认为是终端用户框架，平台

【LSP】

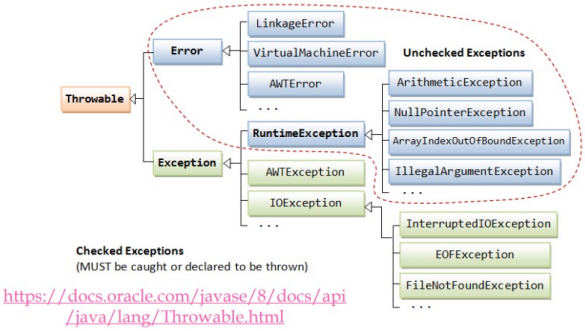
- 里氏替换原则的主要作用就是规范继承时子类的一些书写规则。其主要目的就是保持父类方法不被覆盖。子类的规约变强！
- LSP依赖于以下限制：
  - 前置条件变弱或者不变
  - 后置条件变强或者不变
  - 不变量要保持或增强
  - 子类型方法参数：逆变（规约变强，前置变弱 反着变）
  - 子类型方法的返回值：协变（规约变强，前置变强 协同变化）
  - 异常类型：协变
- 数组是协变的，向上转型是成立的

```
Fruit[] apples=new Apple[size];
```

- 泛型是类型不变的（泛型不是协变的）。举例来说
  - `ArrayList<String>` 是`List<String>`的子类型
  - `List<String>`不是`List<Object>`的子类型
- 类型擦除的结果： <T>被擦除 T变成了Object

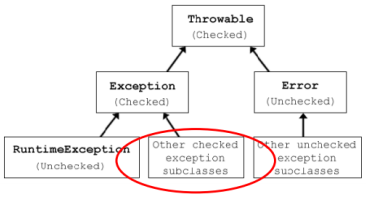
【Throwable】

- 健壮性：输入错误，给出确定的正常的输出，倾向于容错；正确性：出现错误报出错误，倾向于error。
- 对外的接口，倾向于健壮性；对内的实现，倾向于正确性。



https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html

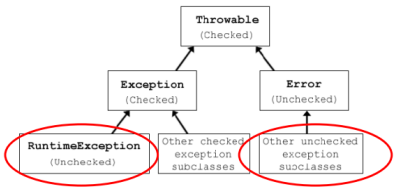
- 运行时异常：由程序员处理不当造成，如空指针、数组越界、类型转换
- 其他异常：程序员无法完全控制的外在问题所导致的，通常为IOE异常，即找不到文件路径等
- checked异常：
  - checked exception是需要强制catch的异常，你在调用这个方法的时候，你如果不catch这个异常，那么编译器就会报错，比如说我们读写文件的时候会catch IOException，执行数据库操作会有SQLException等。



必须捕获并指定错误处理器handler，否则编译无法通过

类似于编程语言中的static type checking

- unchecked异常：
  - 这种异常不是必须需要catch的，你是无法预料的，比如说你在调用一个 list.szie()的时候，如果这个list为null，那么就会报NUllPointerException，而这个异常就是 RuntimeException，也就是UnChecked Exception
  - 常见的unchecked exception：JVM抛出，如空指针、数组越界、数据格式、不合法的参数、不合法的状态、找不到类等



可以不处理，编译没问题，但执行时出现就导致程序失败

类似于编程语言中的dynamic type checking

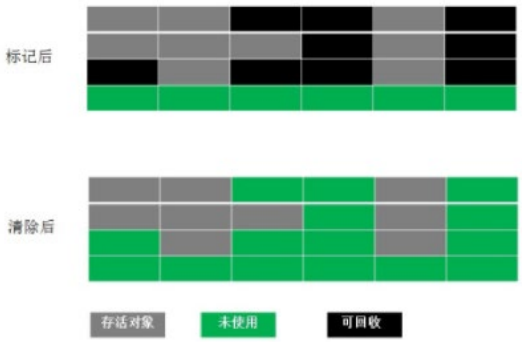
	Checked exception	Unchecked exception
Basic	The compiler checks the checked exception. If we do not handle the checked exception, then the compiler objects. 必须被显式地捕获或者传递（try-catch-finally-throw），否则编译器无法通过	The compiler does not check the Unchecked exception. Even if we do not handle the unchecked exception, the compiler doesn't object.异常可以不必捕获或抛出，编译器不去检查
Class of Exception	Except RuntimeException class, all the child classes of the class "Exception", and the "Error" class and its child classes are Checked Exception. 继承自Exception类	RuntimeException class and its child classes, are Unchecked Exceptions.继承自RuntimeException类
Handling	从异常发生的现场获取详细的信息，利用异常返回的信息来明确操作失败的原因，并加以合理的恢复处理	简单打印异常信息，无法再继续处理
Appearance	代码看起来复杂，正常逻辑代码和异常处理代码混在一起	清晰，简单

【GC的四种策略】

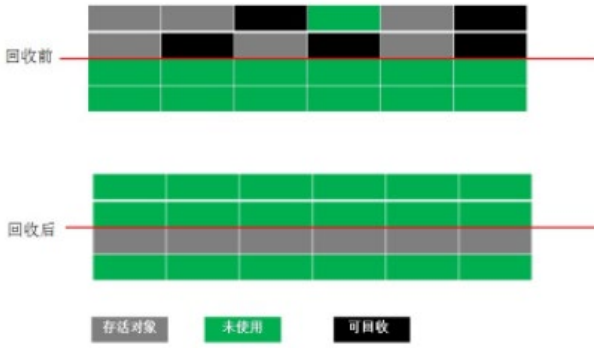
引用计数

- 基本思想：为每个object存储一个计数RC，当有其他 reference指向它时，RC++；当其他reference与其断开时，RC--；如 果RC==0，则回收它。
- 优点：简单、计算代价分散，“幽灵时间”短 为0
- 缺点：不全面（容易漏掉循环引用的对象）、并发支 持较弱、占用额外内存空间、等

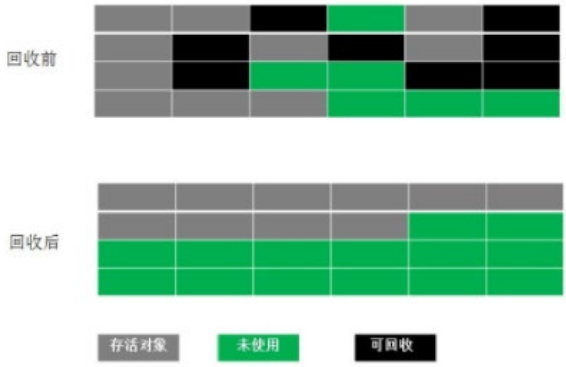
- Mark-Sweep（标记-清除）算法
  - 基本思想：为每个object设定状态位(live/dead)并记录，即mark阶段；将标记为dead的对象进行清理，即sweep可阶段。
  - 优点：可以处理循环调用，指针操作无开销，对象不变
  - 缺点：复杂度为O(heap),高　堆的占用比高时影响性能，容易造成碎片，需要找到root



- Copying（复制）算法
  - 基本思想：为了解决Mark-Sweep算法的缺陷，Copying算法就被提了出来。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。
  - 优势：运行高效、不易产生内存碎片
  - 缺点：复制花费大量的时间，牺牲内存空间



- Mark-Compact（标记-整理）算法
  - 基本思想：为了解决Copying算法的缺陷，充分利用内存空间，提出了Mark-Compact算法。该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。



- 年青一代使用copying算法，年好易贷使用Mark sweep和mark-compact算法

【死锁】

- 产生死锁的四个必要条件：
  - 互斥条件：一个资源每次只能被一个进程使用，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。
  - 请求与保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。
  - 不可剥夺条件:进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。
  - 循环等待条件: 若干进程间形成首尾相接循环等待资源的关系
- 这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。
- 防止死锁的方法：
  - 加锁顺序：当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。如果能确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。这种方式是一种有效的死锁预防机制。但是，这种方式需要你事先知道所有可能会用到的锁，但总有些时候是无法预知的

```
public void friend(Wizard that) {
    Wizard first, second;
    if (this.name.compareTo(that.name) < 0) {
        first = this; second = that;
    } else {
        first = that; second = this;
    }
    synchronized (first) {
        synchronized (second) {
            if (friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

  - 使用粗粒度的锁，用单个锁来监控多个对象
    - 对整个社交网 络设置 一个锁 ， 并且对其任何组成部分的所有操作都在该锁上进行同步。
    - 例如：所有的Wizards都属于一个Castle，可使用 castle 实例的锁
    - 缺点：性能损失大；
      - 如果用一个锁保护大量的可变数据，那么久放弃了同时访问这些数据的能力；
      - 在最糟糕的情况下，程序可能基本上是顺序执行的，丧失了并发性

```
public class Wizard {
    private final Castle castle;
    private final String name;
    private final Set<Wizard> friends;
    ...
    public void friend(Wizard that) {
        synchronized (castle) {
            if (this.friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

【用注释形式撰写测试策略】

```
/**
 * Reverses the end of a string.
 *
 * For example:
 *   reverseEnd("Hello, world", 5)
 *   returns "Hellodlrow ,"
 *
 * With start == 0, reverses the entire text.
 * With start == text.length(), reverses nothing.
 *
 * @param text    non-null String that will have
 *                its end reversed
 * @param start   the index at which the
 *                remainder of the input is
 *                reversed, requires 0 <=
 *                start <= text.length()
 * @return input text with the substring from
 *               start to the end of the string
 *               reversed
 */
static String reverseEnd(String text, int start)
```

Document the strategy at the top of the test class:

```
/*
 * Testing strategy
 *
 * Partition the inputs as follows:
 * text.length(): 0, 1, > 1
 * start:         0, 1, 1 < start < text.length(),
 *                text.length() - 1, text.length()
 * text.length()-start: 0, 1, even > 1, odd > 1
 *
 * Include even- and odd-length reversals because
 * only odd has a middle element that doesn't move.
 *
 * Exhaustive Cartesian coverage of partitions.
 */
```

Each test method should have a comment above it saying how its test case was chosen, i.e. which parts of the partitions it covers:

```
// covers test.length() = 0,
//        start = 0 = text.length(),
//        text.length()-start = 0
@Test public void testEmpty() {
    assertEquals("", reverseEnd("", 0));
}
```

【测试覆盖度】

代码覆盖度：已有的测试用例有多大程度覆盖了被测程序

代码覆盖度越低，测试越不充分但要做到很高的代码覆盖度，需要更多的测试用例，测试代价高

分类：函数覆盖 + 语句覆盖 +分支覆盖 + 条件覆盖 + 路径覆盖

测试效果：路径覆盖>分支覆盖>语句覆盖

测试难度：路径覆盖>分支覆盖>语句覆盖

路径数量巨大，难以全覆盖

【snapshot】在Runtime，code level，moment

<table>
<tr><th></th><th colspan="2">Moment</th><th colspan="2">Period</th></tr>
<tr><th></th><th>Code-level</th><th>Component-level</th><th>Code-level</th><th>Component-level</th></tr>
<tr><td>Build-time</td><td>Source code, AST, Interface-Class-Attribute-Method (Class Diagram)</td><td>Package, Source File, Static Linking, Library, Test Case (Component Diagram) Build Script</td><td>Code Churn (代码变化)</td><td>Configuration Item, Version</td></tr>
<tr><td rowspan="2">Run-time</td><td rowspan="2">Code Snapshot, Memory dump</td><td rowspan="2">Package, Library, Dynamic linking, Configuration, Database, Middleware, Network, Hardware (Deployment Diagram)</td><td>Execution trace</td><td>Event log</td></tr>
<tr><td colspan="2">Procedure Call Graph, Message Graph (Sequence Diagram)

Parallel and multi-threads/processes
Distributed processes</td></tr>
</table>

【SOLID】

设计模式前五个原则，恰恰是告诉我们用抽象构建框架，用实现扩展细节的注意事项而已：

单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原则是总纲（实现效果），它告诉我们要对扩展开放，对修改关闭。