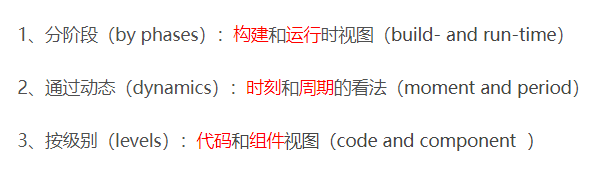
1. 软件构造的多维视图



Build-time views 简单翻译为构造阶段视图，从想法到用户需求到产品设计到代码编写再到可安装/可执行的程序包，其实都属于构造阶段。

从code-level来看，build-time views意味着代码的逻辑组织，源代码是如何通过例如functions, classes, methods, interfaces等基本的代码块有逻辑的组织起来，以及相互之间的依赖关系。

从component-level来看，build-time views意味着代码的物理组织，源代码是如何通过 files, directories, packages, libraries组织起来，以及相互之间的依赖关系。

从moment-view来看，build-time views意味着特定时刻的软件形态。

从period-view来看，build-time views意味着软件形态随时间的变化（一段时间的变化）

Run-time views 简单翻译为运行阶段视图，当你的程序在目标机器中运行时表现如何，以及在运行时目标机器要将哪些磁盘文件存入内存等等，这些问题都是run-time views所关注的。 其实就是编译阶段和运行阶段的区别

从code-level来看，run-time views意味着一个可执行程序在内存中的状态以及程序内部各个单元间（例如Objects和functions等）如何相互调用。

从component-level来看，run-time views意味着软件包是如何部署到具体的物理环境中（例如操作系统，网络，硬件等），以及他们之间如何相互作用。

从moment-view来看，run-time views意味着一个程序在特定时刻的表现。

从period-view来看，run-time views意味着程序随时间变化的一系列表现，包括过程调用图（Procedure Call Graph），消息图（序列图）（ Message Graph (Sequence Diagram)），并行和多线程/进程（Parallel and multithreads/processes），分布式进程（Distributed processes）等。

**（1）Build-time, moment, and code-level view**

词汇层面指的是基于词汇的半结构化源代码（Lexical-based semi-structured source code），半结构化是指近乎自然语言的风格+遵循特定的编程语法，前者方便程序员，后者方便编译器。

语法层面指的是面向语法的程序结构（Syntax-oriented program structure），具体例子有抽象语法树（AST）

语义层面指的是面向语义的程序结构（Semantics-oriented program structure），语义是指源代码具体想实现什么目标。具体例子有使用类图Class Diagram（UML）来描述 interfaces, classes, attributes, methods以及它们之间的关系。下图既是一张Class Diagram。



### （2）Build-time, period, and code-level view

这个多维视图描述了代码随时间的变化（Code churn），包括代码行的增加，删除和修改。

### （3）Build-time, moment, and component-level view

这个多维视图着眼于源代码被物理地组织成文件（file），这些文件进一步由目录组织。文件被封装成包（package），并在逻辑上封装组件和子系统。可重用模块以库（library）的形式存在

在编程时和build时，需告诉IDE和JVM在哪里寻找某些库 ，将库整合到可执行程序中有两种不同方法，静态链接和动态链接，静态链接发生在build-time阶段

静态链接

### （4）Build-time, period, and component-level view

这个多维视图着眼于各项软件实体（例如files/packages/components/libraries）随时间如何变化，Software Configuration Item (SCI，软件配置项) 以及版本（Version）的演变

我们通常使用Version Control System (VCS)即版本控制系统来对软件的版本进行管理，现在流行的VCS主要是git

### (5) Run-time, moment, and code-level view

这个多维视图着眼于程序运行时在某个特定时刻的代码层面的状态，主要工具有快照图（Snapshot diagram）和内存信息转储（Memory dump ）。

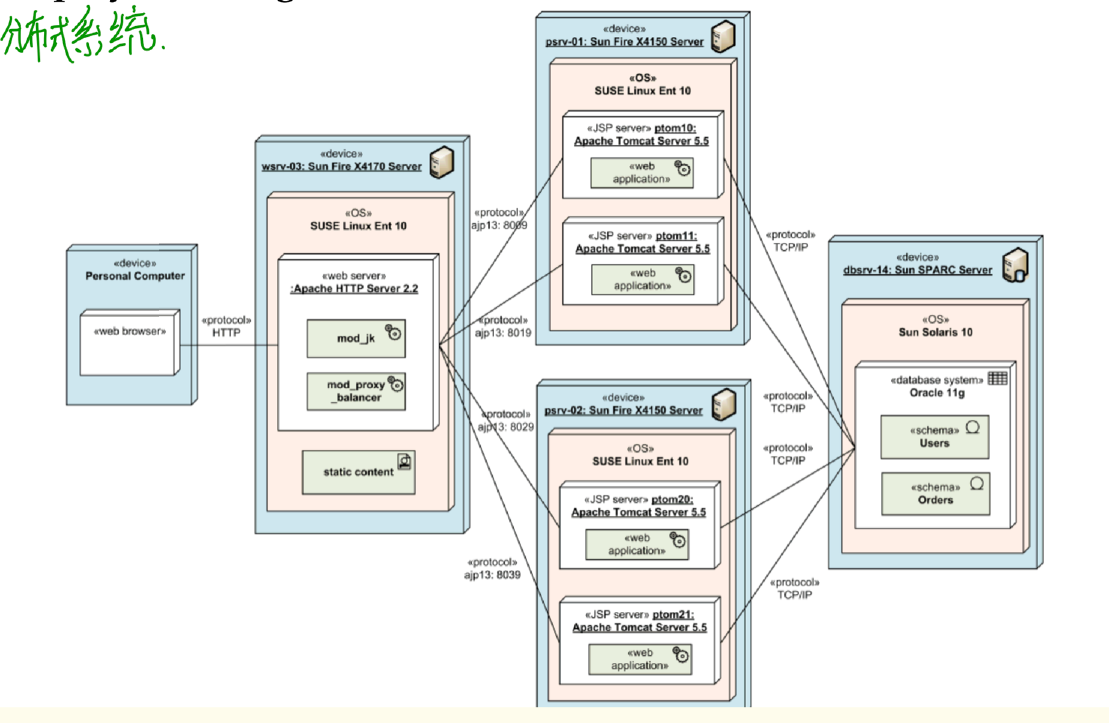
### (6) Run-time, period and code-level view

这个多维视图着眼于程序运行阶段在某一段时间内在代码层面的视图，以下面UML中的Sequence diagram为例，图中体现了程序各个单元间的交互。

调用关系 程序的 执行追踪和堆栈追踪

### (7) Run-time, moment, and component-level view

这一多维视图着眼于程序运行阶段在某一特定时刻在组件层面的视图，以下面UML中的部署图为例，图中展示了各个组件的相互联系。



### (8) Run-time, period, and component-level view

这一多维视图着眼于程序在运行阶段在某一阶段的组件层面视图，这一视图主要涉及到事件日志（Event logging），事件日志为系统管理员提供了对诊断和审计有用的信息。每一类事件都被赋予一个唯一的“代码”来格式化并输出一条人类可读的消息。这有利于本地化，并允许系统管理员更轻松地获得有关发生问题的信息。

AST树 对代码进行结构化，对树的修改就是对代码的修改

Execution tracing 代码层面

第三章

软件构造内部和外部的质量指标

外部质量指标 对于用户来说。

正确性 能够按照之前订立的规约来执行 确保正确性

1. 测试和程序运行的调试
2. .防御式编程
3. .形式化方法

健壮性 即程序针对异常情况出现时的处理。

可拓展性 对软件规约的修改是否容易

可复用性

兼容性 不同的系统之间可以集成

1. 需要保持设计的同构性
2. 标准化 standardized file format

standardized data structures

standardized user interfaces

standardized access protools

性能 性能的优化要确保正确性还要考虑安全性因此对性能的优化要对各方面进行折中

Portability 可移植性

Easy to use

Functionality

**软件构造的内部指标**

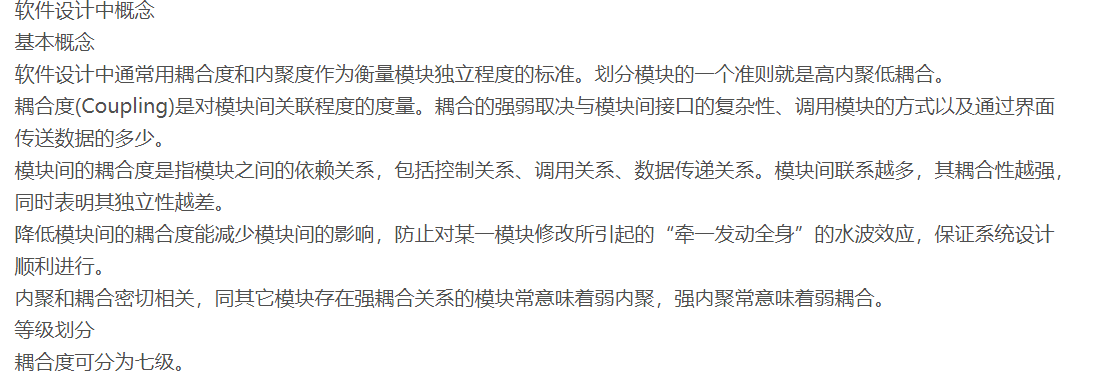
与源代码相关 lines of code LOC

Cyclomatic Complexity(圈复杂度，数量上表现为独立现行路径条数，即合理的预防错误所需测试的最少路径条数，圈复杂度大说明程序代码可能质量低且难于测试和维护)

与结构相关 1.耦合度

2.聚合度

是指对于软件内部的 实现模块来说的 聚合度高的模块独立性高 。耦合度是指两个模块之间的联系的程度



此外 还需要注意代码的可读性 代码的数量的大小

软件配置管理SCM与版本控制中心VCS

软配置管理SCM

追踪和控制软件的变化 包括：

1. revision 修改 control 2. Establishment of baselines

SCI(软件配置项) 软件中发生变化的基本单位

Baseline 软件持续变化过程中的稳定时刻，我们对外发布的版本。

CMDB

配置管理数据库 储存物件的各配置项随时间发生变化的信息+基础

版本控制系统VCS

对个人开发者来说

可以回复到上一个版本

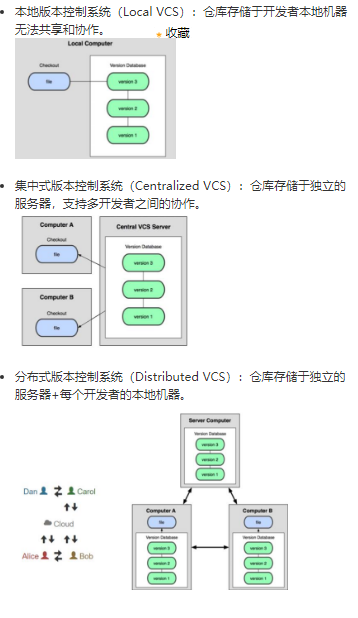
可以比较两个版本之间的差异

备份软件版本的历史

获取备份

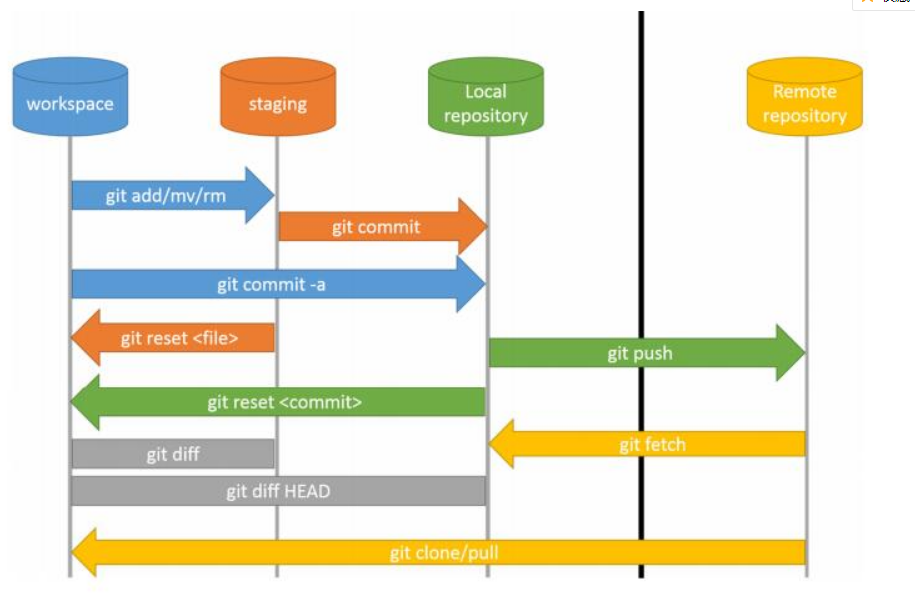
对于开发团队 可以完成在多个开发者之间的共享和协作 记录每个开发者的动作，便于审计

版本控制系统的分类 本地版本控制系统 集中式版本控制系统 分布式版本控制系统



Git 和GitHub





Version Control System，简称VCS，属于软件配置管理（SCM）的一个部分

第三章

基本数据类型 ： 只有值没有ID 这里我理解ID为一个指针

在栈中分配内存 不可变 代价比较低 一般用小写表示

对象数据类型 有ID 开头的第一个字母一般要大写 有些可变 有些不可变

定义一个代价较为昂贵

1．静态类型检查 在程序运行前自动发现bug 在编译时发现问题

检查范围 语法错误 类名/函数名错误(违法java内部命名规则)

参数的数目不对应 参数的类型不对应 参数返回值错误

与类型相关

2.动态类型检查 在程序运行时发现bug

非法的参数值（例如/0的错误） 非法的返回值 数组越界 空指针错误

与值相关

**Mutable/Immutable**

**可变性**

**改变一个变量 将变量指向另一个存储空间**

改变一个变量的值 向该变量当前指向值的存储空间中添加一个新值

Immutability 不变性

不能被改变的数据类型 一旦被创建其值再不能被改变

引用类型 一旦确定指向对象 就不可以改变

Final关键字 若java编译器不能确定final修饰的变量会不会改变就会报错，这也属于静态检查

Final修饰类 无法再产生子类 修饰变量 无法改变其值或者引用 修饰方法 则该方法不能被重写

可变对象拥有一个可以改变自己的值或者引用的方法，而不可变对象没有

可变类型的有点

1. 使用不可变类型，需要进行大量的临时拷贝
2. 可变类型可以最小化拷贝以提高效率
3. 可以获得更好的性能
4. 适合于在多个模块之间的共享数据

不可变类型 更安全，更好理解，对改变的支持也更好

常见可变类型 StringBuilder,Date,List<>

常见的不可变类型 LocalDateTime Instant 基本类型及其封装类型

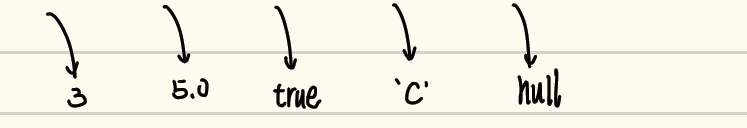
UnmodifiableList unmodifiableSet unmodifiableMap 不可改变

它们接收一个List Set Map ，包装为一个不可变List 如 List<String>

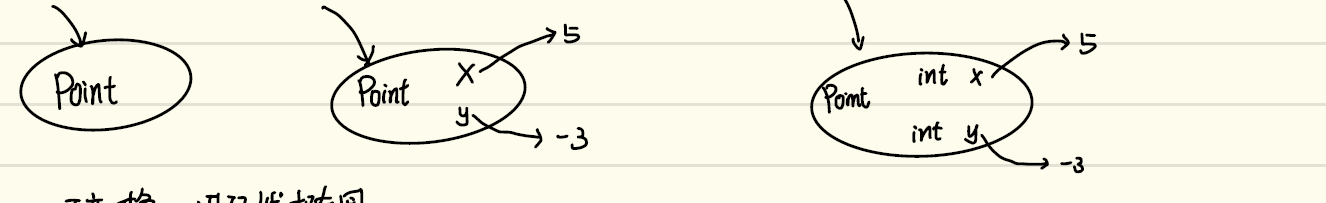
防御式拷贝

Snapshot diagram快照图 runtime code-level moment

基本类型的值

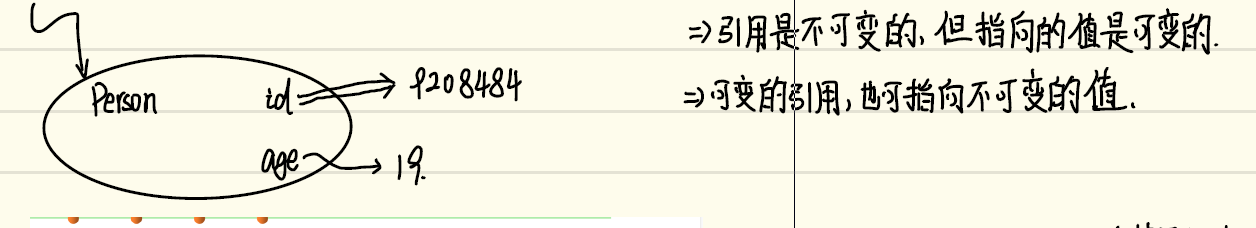


对象类型的值 一个椭圆圈

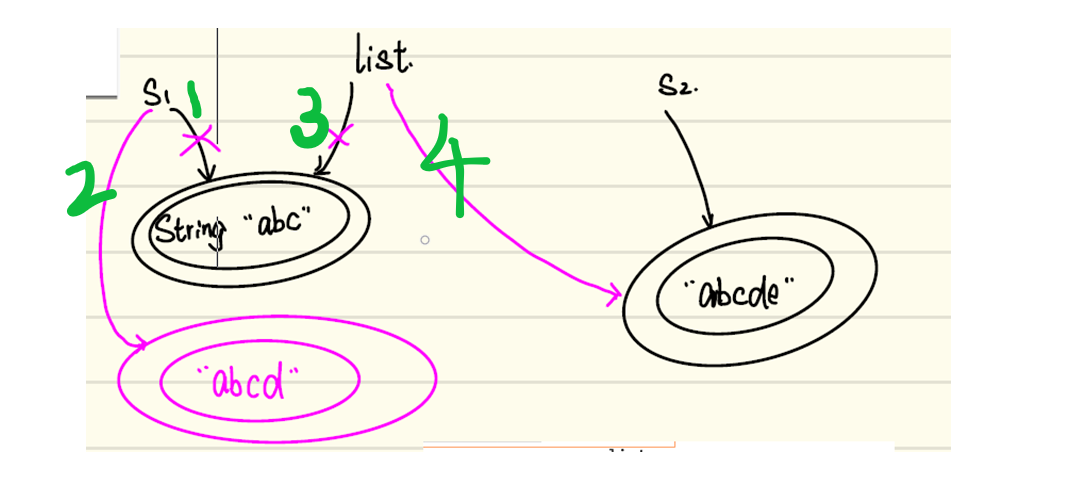


不可变对象 用双线椭圆圈表示

不可变引用 用双线箭头表示







Iterator对一个List进行遍历的时候，不可以改变List之中的元素，即remove add 等，否则编译器在运行时会报错

即不可对原数组进行直接操作，但是可以使用iterator自带的remove函数

Specification简写为spec 前置/后置条件

Spec

1.程序与客户端之间达成的一致

2.调用时供需双方都有责任遵守spec的规定

3.只要满足spec无需通知客户端就可以对程序修改，隔离了变化

.4. 用户不知道实现细节 程序员不知道用法细节 ，二者在满足spec的基础上可以任意修改代码

.spec包括 输入和输出的数据类型，程序要实现的功能和正确性 性能等指标

前置条件 对客户端的约束，使用方法时必须满足的条件 @parameter

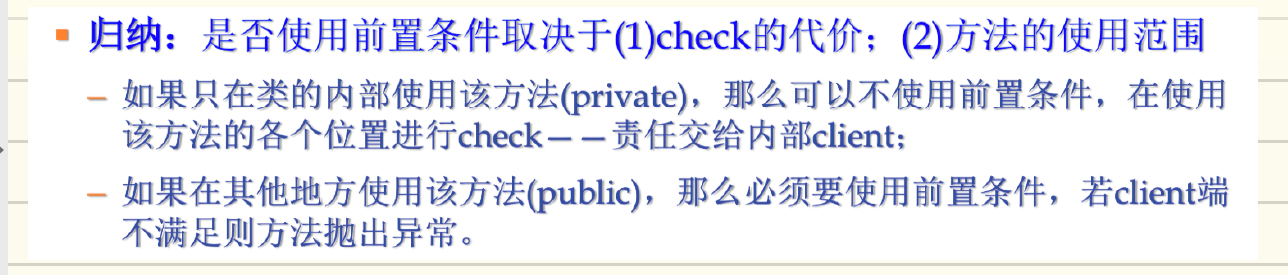
后置条件 对开发者的约束 方法结束时必须满足的条件

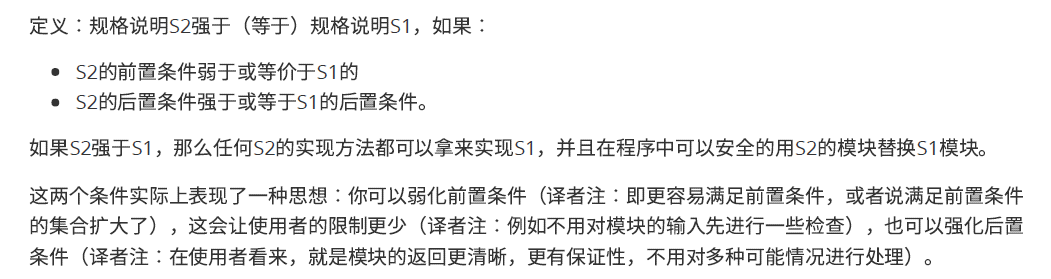
契约就是前置条件满足了那么后置条件必须被满足 、

注意 no mutation of inputs原则 对于multating method 除非在后置条件中明确说明，否则方法内部不允许修改输入的参数。

笼统的说，规格说明分为两种：操作性的（Operational）规格说明给出了实现过程的步骤就像伪代码⼀样，⽽声明性的（Declarative）规格说明不对实现过程进⾏要求，它们仅仅给出最后输出的属性和意义，以及它们和输⼊之间的关系。在绝⼤多是情况下，声明性的规格说明更合适。它们通常会更简洁、更易懂、并且最重要的是，它们不会让使⽤者

尝试依赖特定的实现⽅案（很多时候⼀个模块的实现⽅案会不得不改变）。例如，如果我们想要允许多种⽅案来实现find ，我们就不会在规格说明要求“从数组低位开始向上遍历搜索”。有些时候，程序员想要给维护者（maintainer）模块的实现信息，于是他们将实现描述写在了规格说明中。要记住，规格说明是给使⽤者⽽⾮模块的开发者使⽤的，如果你想要⽤描述模块的实现⽅法，将它们注释在模块⾥⾯。

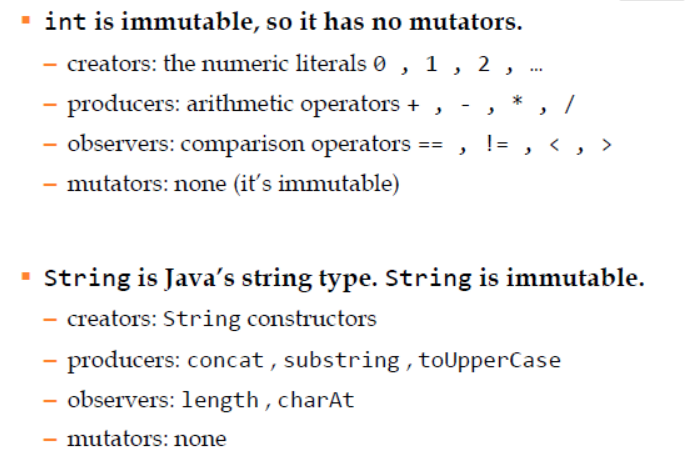


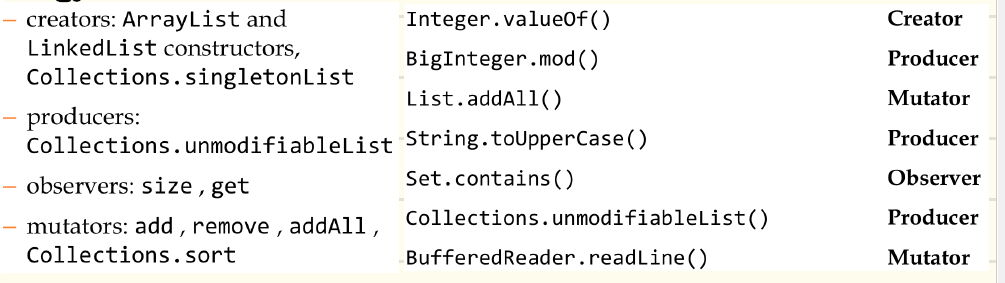


比较后置条件的时候 注意要保证前置条件相同。

ADT操作的四种类型

* 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
  + 例子：Integer.valueOf( )
* Producers（生产器）：
  + 通过接受同类型的对象创建新的对象。
  + T+ , t\* -> T
  + 例子：String.concat( )
* 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
  + 例子：List.size( ) ；
* 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
  + 例子：List.add( )
* 解释：T是ADT本身；t是其他类型；+ 表示这个类型可能出现一次或多次；\* 表示可能出现0次或多次。





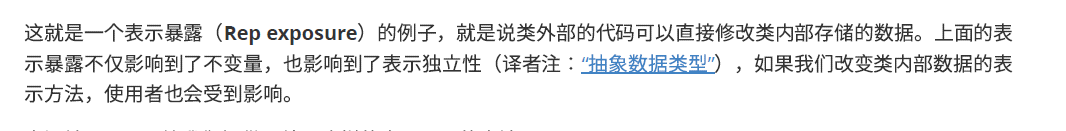


ReadLine 需要特殊记忆 它有返回值

表示独立性

用户使用ADT时没有必要考虑ADT内部如何实现，ADT内部表示的变化不会影响spec和客户端

表示泄露

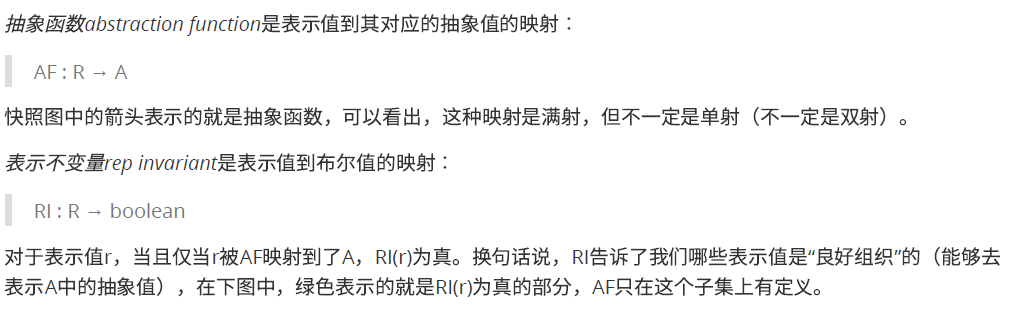


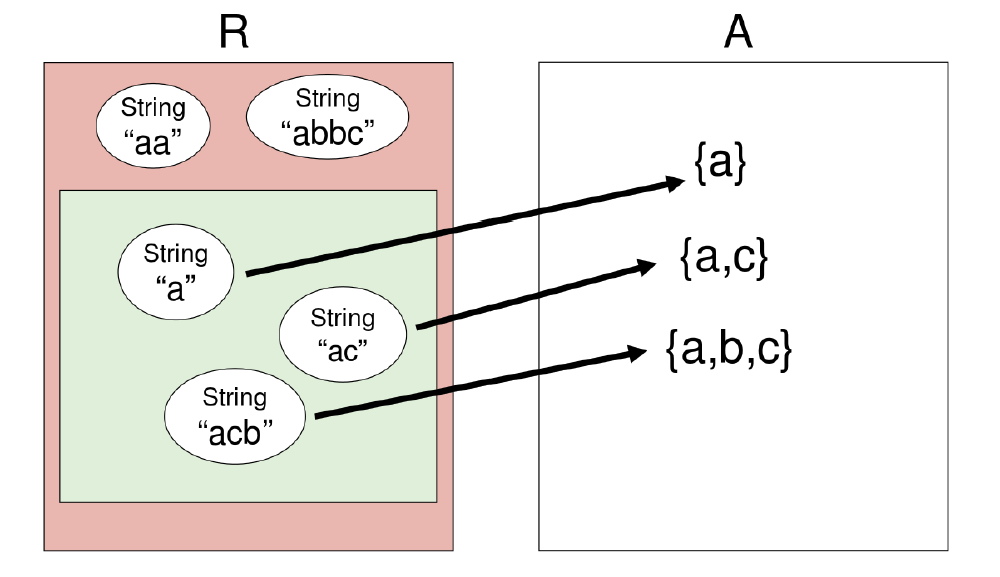
表示不变量 RI

不变量 ADT中无论如何都要成立的部分 由ADT来完全负责自己的表示不变量， 使用表示不变量来检查程序的正确性

使用private final 来保证RI不变量的方法

使用防御性copy 返回mutable类型时 将该内容copy到一个新的对象中返回





R 是所有真正实现的集合

A 是R 中所有对象试图表达的东西 是字符的集合

它俩之间的映射关系 就是 AF

必须是满射 A中的每一个元素都能在R中找到对应

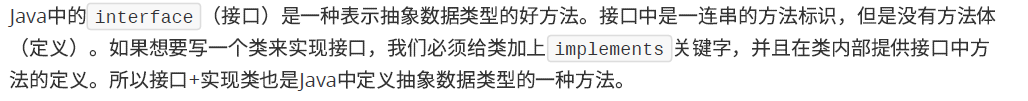
未必是单射

以注释的形式写AF/RI

应该包括 RI AF Safe from rep exposure 三部分

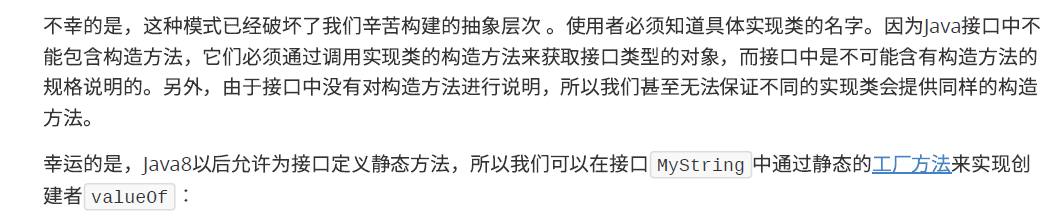
ADT的spec中不能有内部表示的细节或者任何R中的值 可以提到A的值 只能用用户可见的内容来撰写 包括参数返回值 异常等

接口



接口之间可以相互继承 一个类可以实现多个接口 一个接口也可以有多个实现类

接口和实现的类在使用的时候必须知道对应的实现该接口的名称



使用静态工厂方法来返回一个特定类型的值 避免破坏抽象层次

抽象类

抽象方法 一个只有方法的声明的方法 用abstract关键字表示

抽象类 至少包含一个抽象方法的类

接口 只含有抽象方法的抽象类

Override

Rewriteable Methods 可以被重新实现的方法

子类中只能添加新的方法 但是无法重写超类中的方法

Java中不能被重写的方法需要在声明中加入final关键字

Final field 代码初始化之后无法被重新赋值

Final method 防止这个方法被重写

Final class 防止类的继承

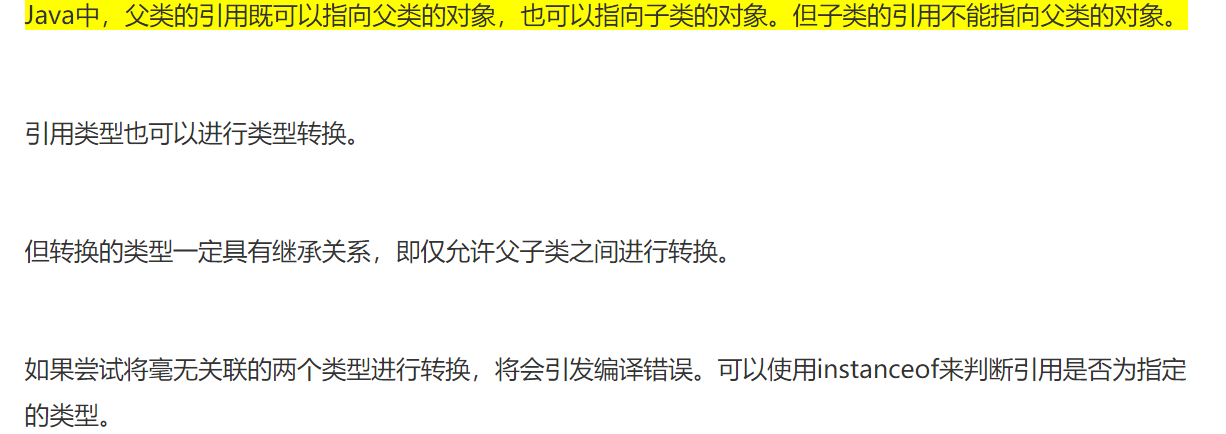
重写函数与原函数有一样的函数声明super可以调用父类函数的功能 在构造函数中super只能放在第一排

多态 overload

多态的三种类型

1 特殊多态 功能重载

2 参数多态 3 子类型多态，包含多态



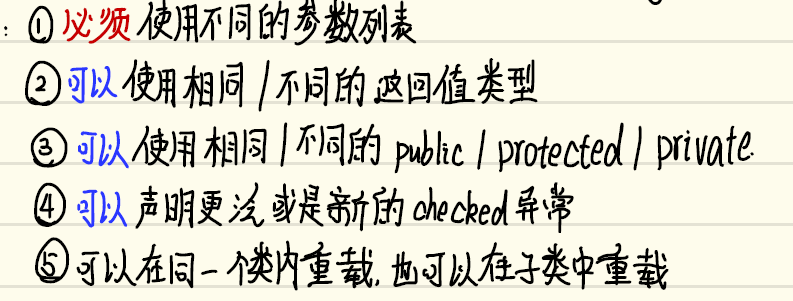
也就是说 所有可以用父类型的地方使用子类型一定不会报错

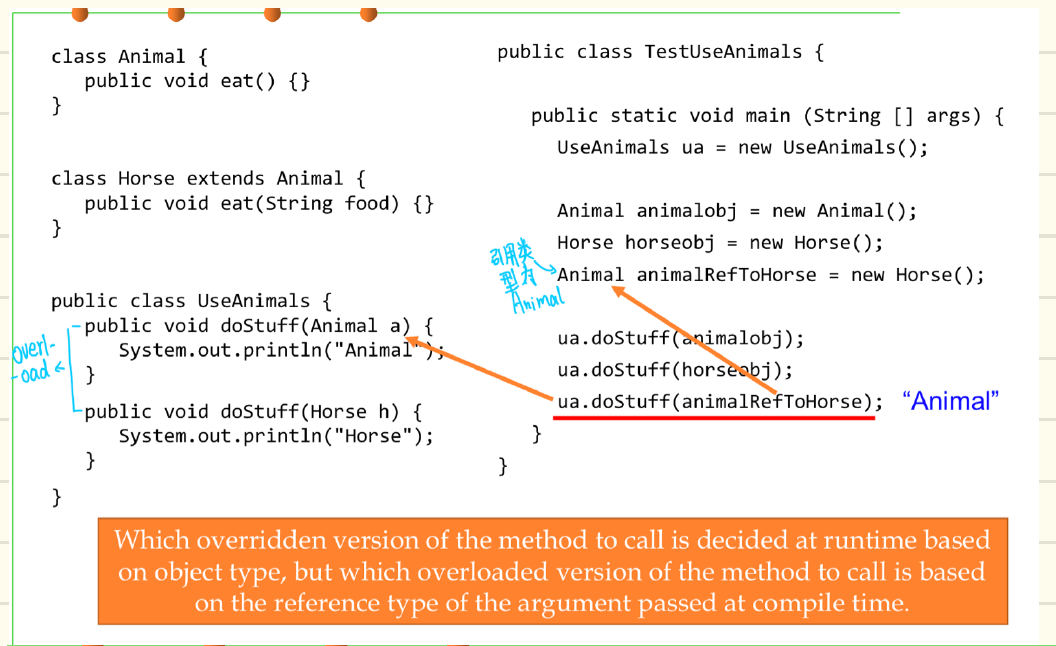
注意 overload 只需要方法名相同 而override则需要整个方法的声明都与父类相同

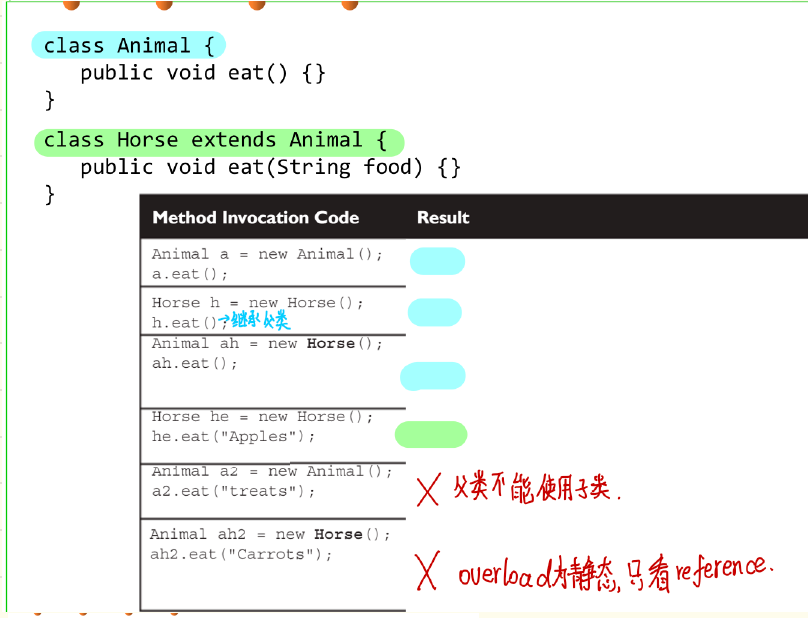
Overloading 多个方法具有相同名字 但有不同的参数列表或者返回值类型

方便用户调用，用户可以用不同的参数列表来调用名称相同但是内涵不同的函数

重载是一个静态的多态 根据参数列表来选择执行的方法，需要进行静态类型检查 在编译阶段决定具体要执行哪种方法







子类重载的函数 父类是可以调用的

泛型

泛型变量 使用< > 来定义的变量

泛型类 定义中包含了泛型变量的类 同理有泛型接口和泛型方法

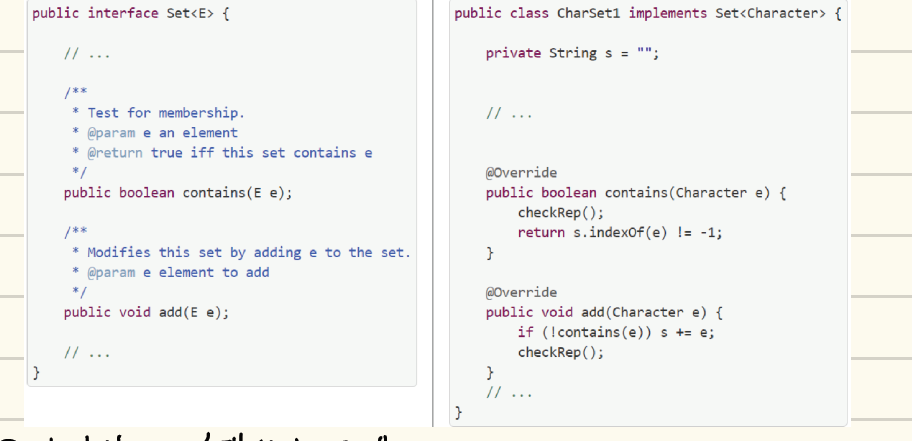
可以使用多个泛型如Map<E,F>

可以使用通配符？只在使用泛型的时候出现 但不能在定义中出现

编译的时候会被类型变量取代 而且泛型不存在泛型数组

泛型接口有两种实现形式

非泛型的实现 泛型的实现



Public interface Set<E> public HashSet<E> imolements Set<E>{

{   
} }

子类型多态 B是A的子类型 得到每个A都是B

B的spec至少跟A一样强

子类型多态 不同类型的对象可以统一的处理无需区分

LSP 若S为T的子类，

等价

Equals的自反，传递，对称

等价的三种定义方式

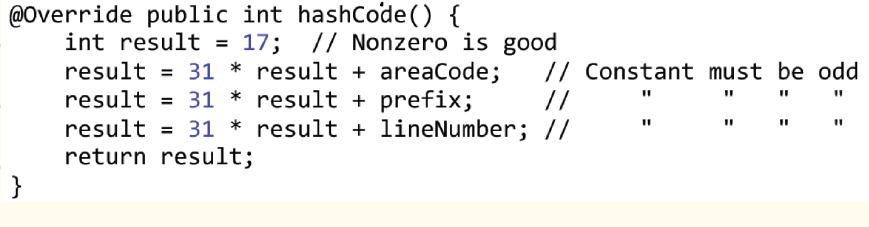
1. 若 AF映射到同样的结果，则等价
2. 若两个对象之间满足自反传递对称的关系，那么为等价关系
3. 站在外部观察者角度发展二者没有区别
4. 站在外部观察者的角度发现二者没有区别

对于ADT来说，“观察”就表示调用该类的运算符

1. 除非对象被修改 否则多次调用该对象得到的结果应该是一样的
2. Hashcode() 返回结果必须一致 相等的对象
3. A.equal(null) 一定返回false

Hashcode 和equals 一起被重写

17 31 法则 这两个比较大的质数



可变观察对象的观察等价性和行为等价性

Object中缺省使用“==”来判断两个对象是否相同

Instanceof操作应该只在equals方法内使用

观察等价性，在不改变状态的情况下，两个mutable对象是否看起来一致

调用observer producer creater 对比两者的状态

行为等价性，调用对象的任意方法，都展现出一致的结果

对于可变对象来说往往倾向于实现严格的观察等价性

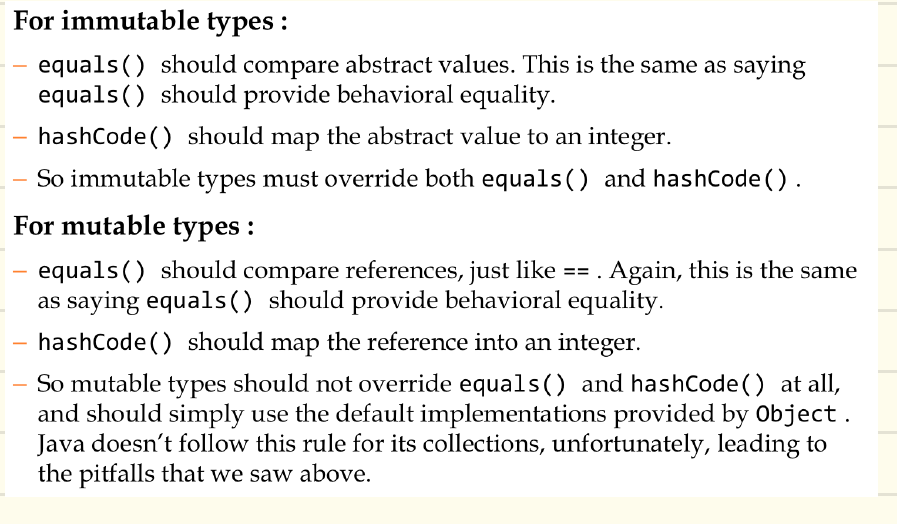
有时候观察等价性可能会导致bug

如果某个mutable对象包含在集合类当中当其发生改编后集合类的行为不确定

Collection 使用观察等价性

Stringbuilder 使用行为等价性

对于mutable对象无需重写hashcode() equals()保留缺省值即可



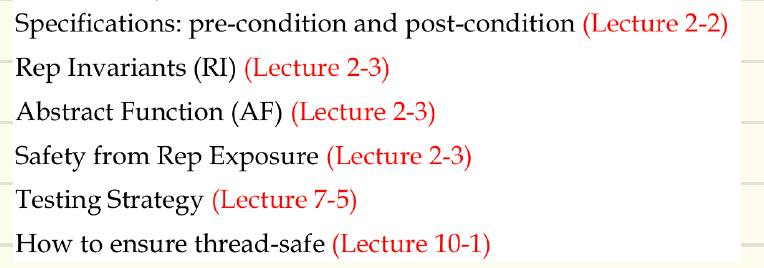
注释的种类

1 title comments

2 introductory comments

3 black comments 4 sigle-Line/Tracimg/End-of-line comments

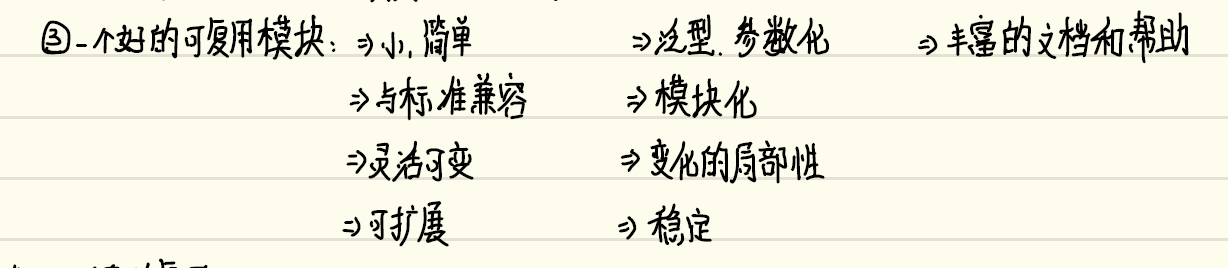
ADT的注释 应当包括



编码规范 怎样正确排序引用的

1 Stabdard packages(java.io) 2 third party packages 3 your own packages

可复用型编程



复用的层次 最主要是代码的复用

软件构造过程中任何实体都可能被复用

白盒复用 源代码可见 可修改和扩展 可定制化程度高 需要对内部有充分的了解

黑盒复用 源代码不可见 简单清晰 但是适应性较差

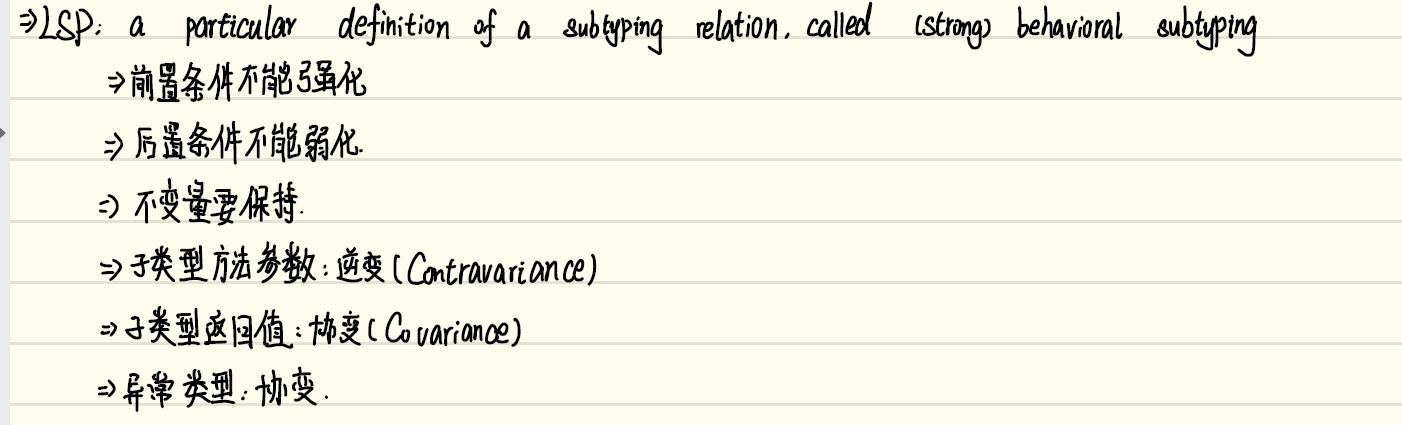
LSP子类型多态 客户端可以用统一的方法处理不同类型的对象

子类型可以增加方法 但是不可以删除方法

子类型要实现抽象类型中所有未实现的方法

子类型中重写的方法必须要求使用同样类型的参数并且返回同样类型的值而且这个重写的方法不能抛出新的异常

更强的不变量 更弱的前置条件 更强的后置条件



协变 父类型->子类型 ：越来越具体 返回值类型不变或者变得更具体

异常类型不变或者变的更具体

反协变 父类型到子类型 越来越抽象 参数类型不变或者越来越抽象

数组是协变的对于所有T类型的数组 可以接收T的所有子类

如果子类只需要父类中的一小部分方法可以选择不使用继承而使用委派来实现

委派的类型

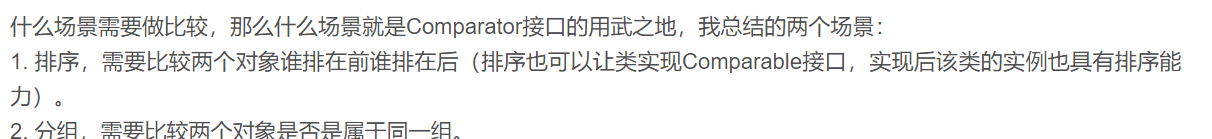
1 临时性 作为某个方法的参数放到List里面

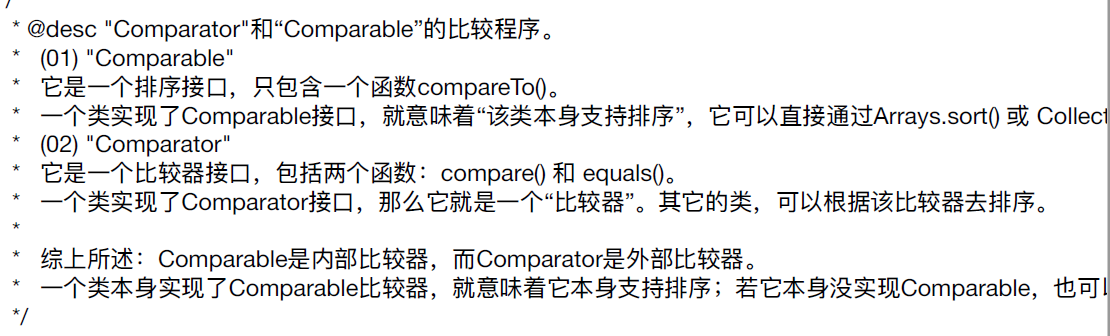
2 永久性 作为类里面的一个属性或者变量

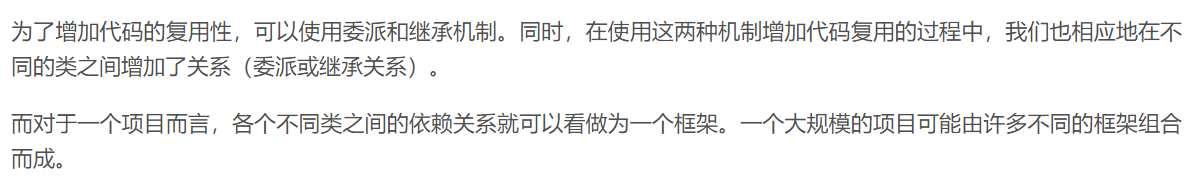
3 更强的 委派 作为某一部分

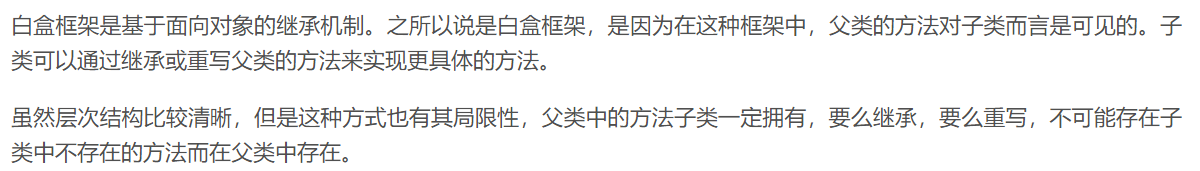
Aggregation 将delegation的类放到构造函数中

Comparator 实现该接口意味着该类支持排序

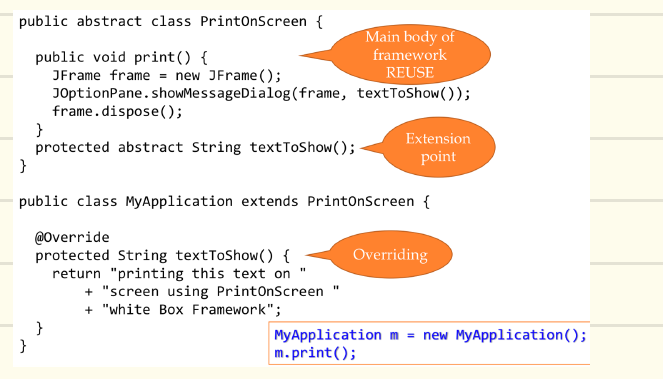


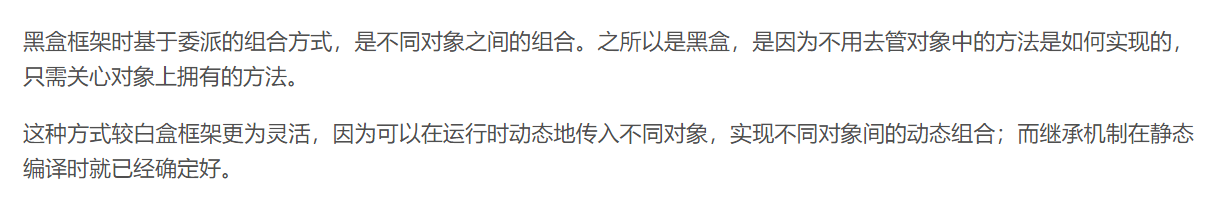


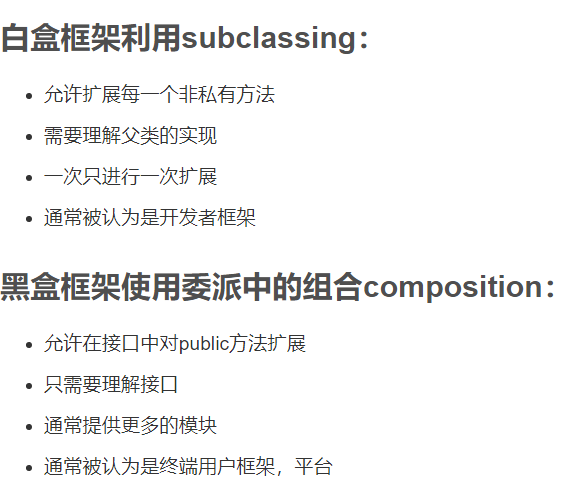










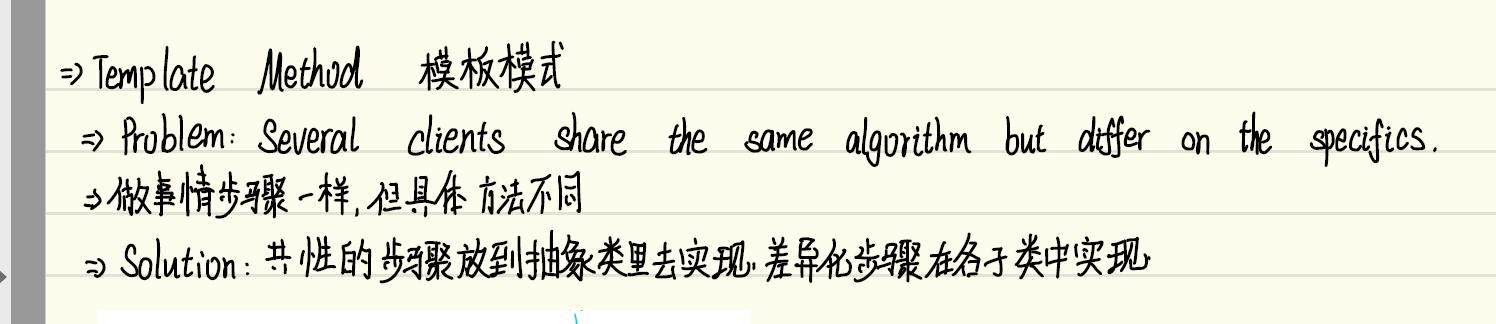


Java中的设计模式

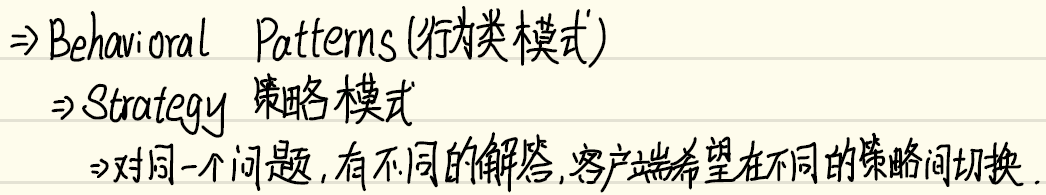
结构设计模式

1. Adapter适配器模式

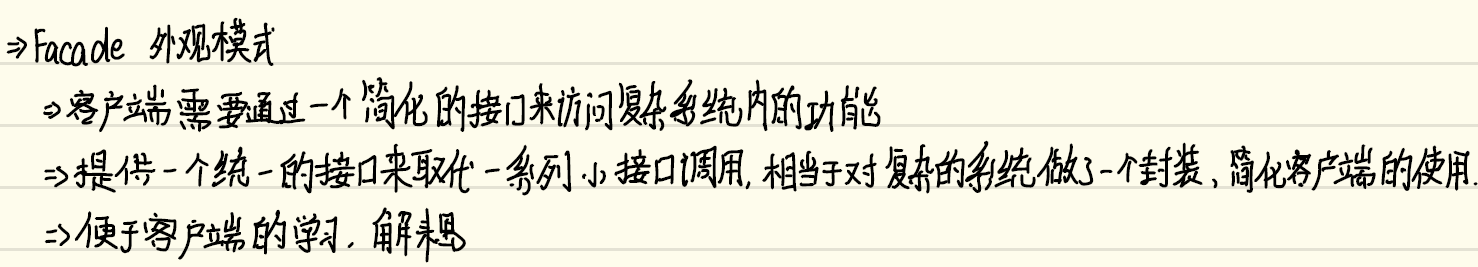
将某个类/接口转换为用户期望的其他形式 增加一个接口将已经存在的子类封装卡里 从而能够隐藏具体子类 通过修改接口让无法合作的类进行合作

2.

在抽象类型中可以使用final关键字对共性的方法进行锁定防止之后重写

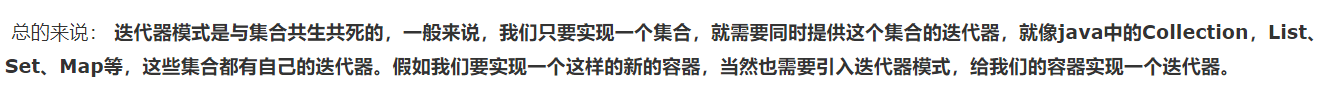
3. 

4.

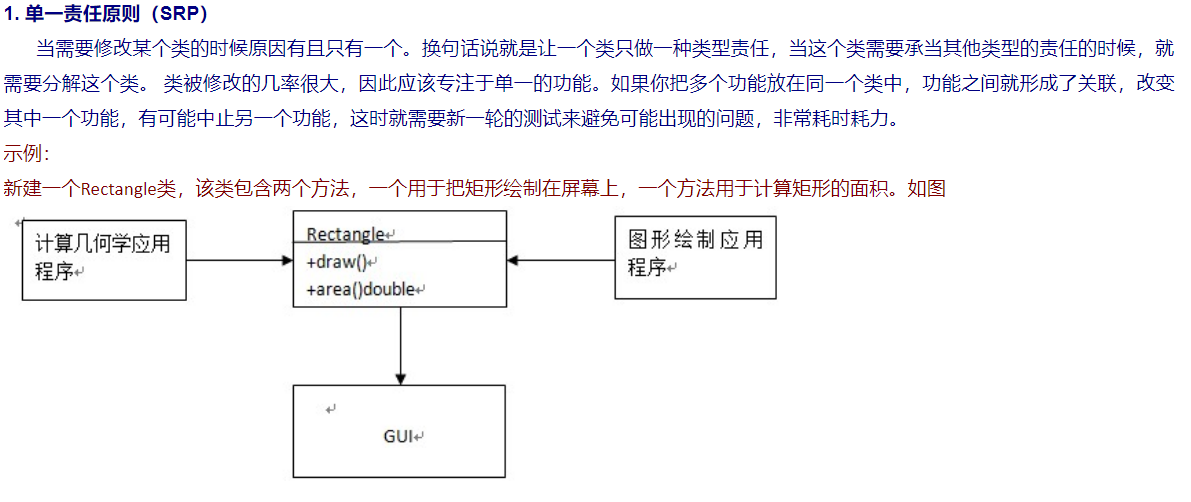


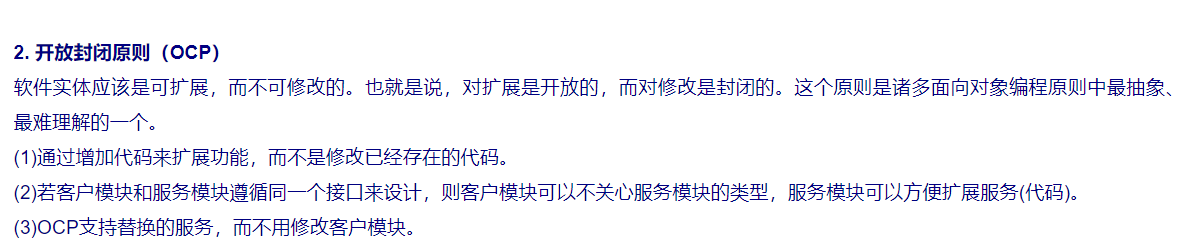
5.

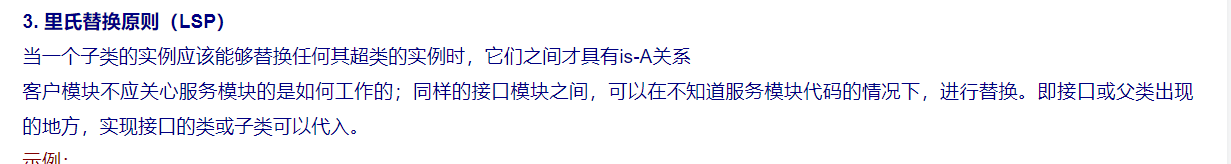


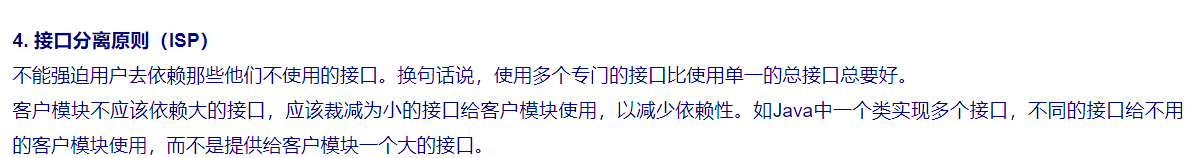


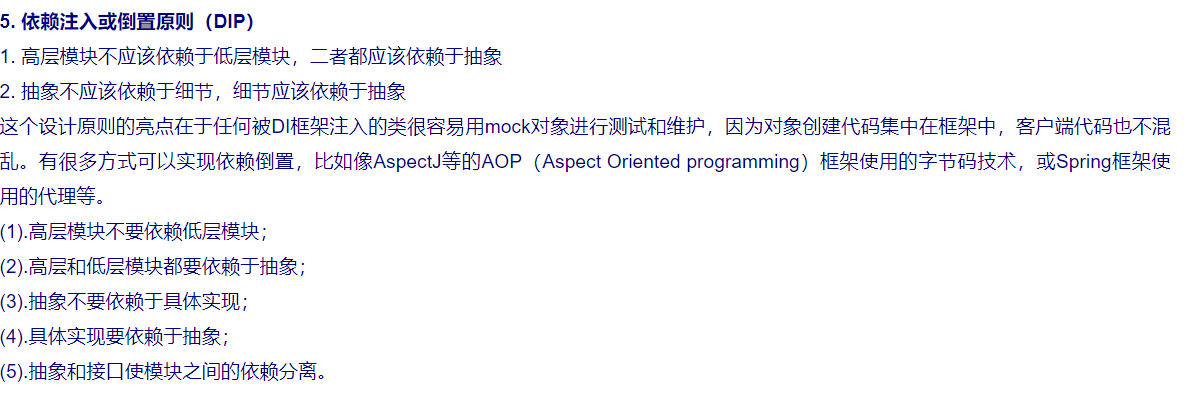












单一责任原则SRP 开放封闭原则OCP open close

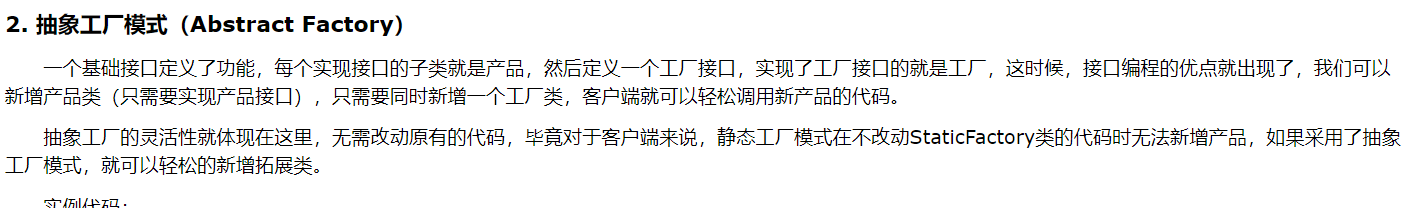
里氏替换原则LSP 接口分离规则 ISP 依赖注入或倒置原则DIP

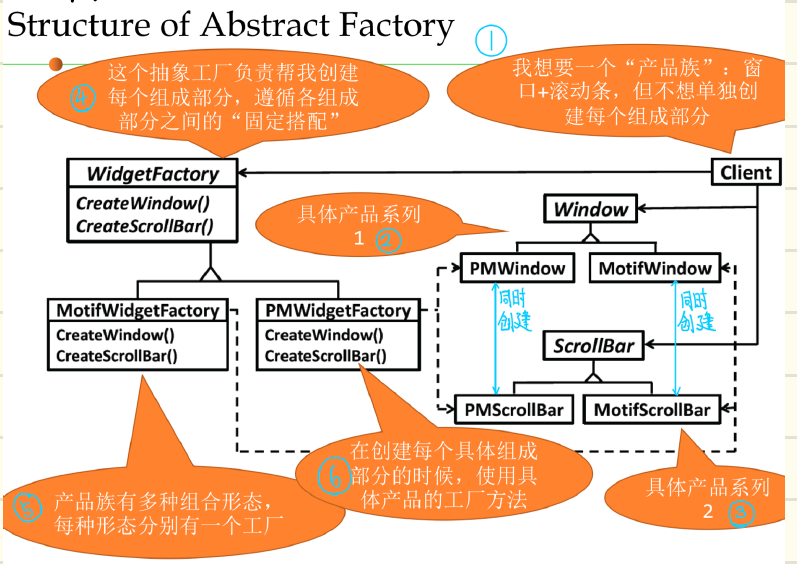
高层和底层模块都应该依赖抽象

Creation Patterns

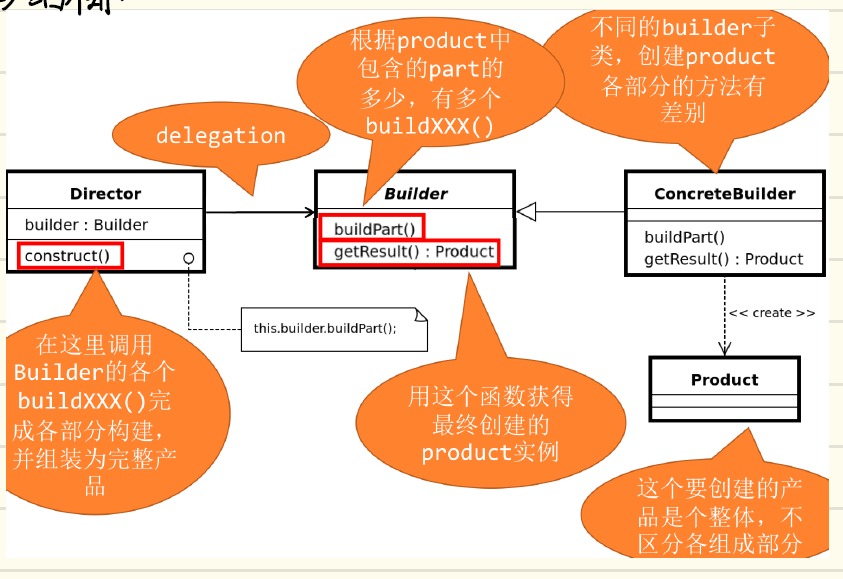
工厂模式 用户不知道要创建哪个类的具体实例

定义一个用来创建具体对象的接口让它的子类来决定实例化哪个类





Build模式





程序的健壮性和正确性

健壮性 输入不正常 仍能表现正常 能运行的程度

处理没有期望到的行为和错误导致的程序中止

即使程序中止，也要向用户准确无歧义的全面展示错误信息

正确性 永不给用户错误的结果

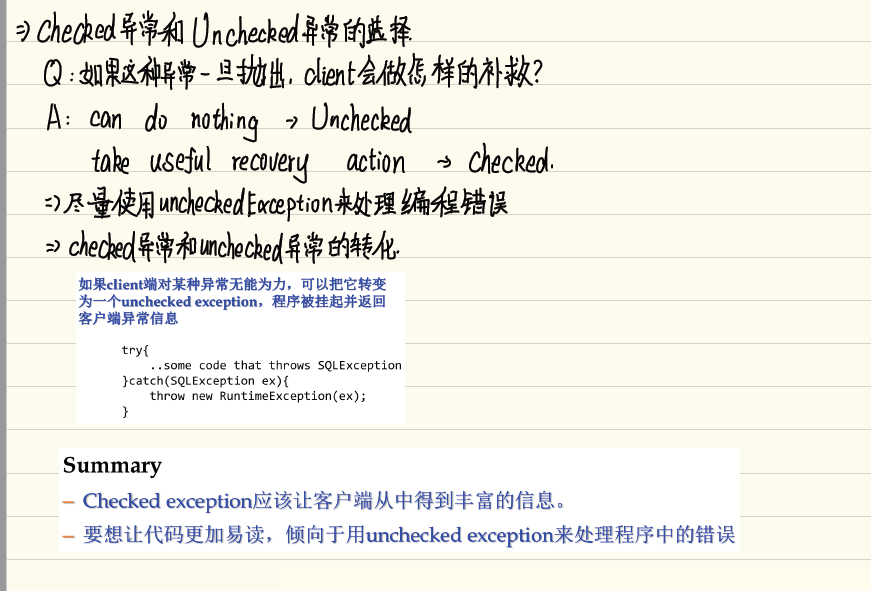
异常

Throwable java中所有的子类都是从throwable中发展出来的子类

Runtime 异常 程序员在编程时的错误导致的

如 数组越界 空指针错误

其他异常： 由外部原因造成的 I/O异常 打开一个不存在的文件

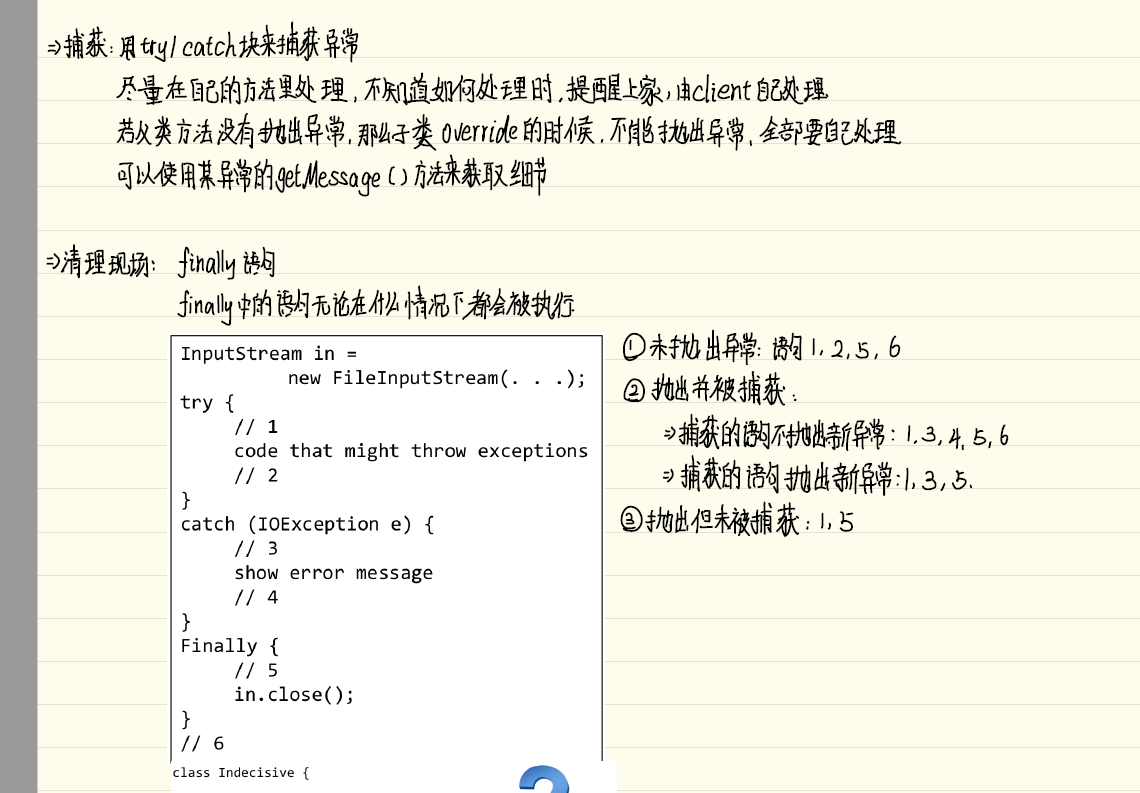


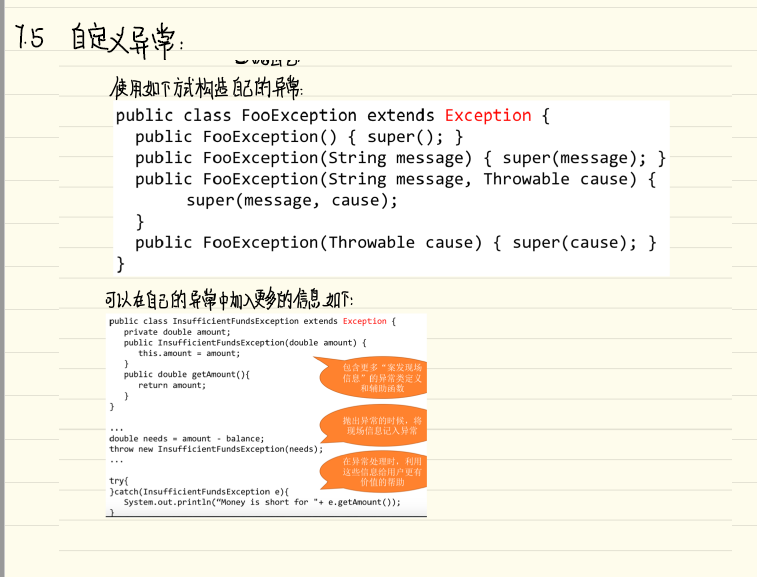
Checked 用户可以进行操作属于可预料但不可预防的异常

异常属于spec的一部分 应该写在Post-condition当中

子类不能抛出比父类更广阔的异常 override 协变

父类型没有抛出异常 那么子类也不能抛出异常





断言 Assert 在开发阶段的代码中嵌入 检查某些假设是否成立 若成立，表明程序正常 如果断言中的条件不被满足那么会抛出AssertionError

断言可应用于： 内部不变量 表示不变量 控制流不变量 方法的前置条件（检查方法的参数） 方法的后置条件（检查方法的返回值）

控制流不变量 若if switch语句没有覆盖所有的可能 就用assert语句来覆盖不合法的情况

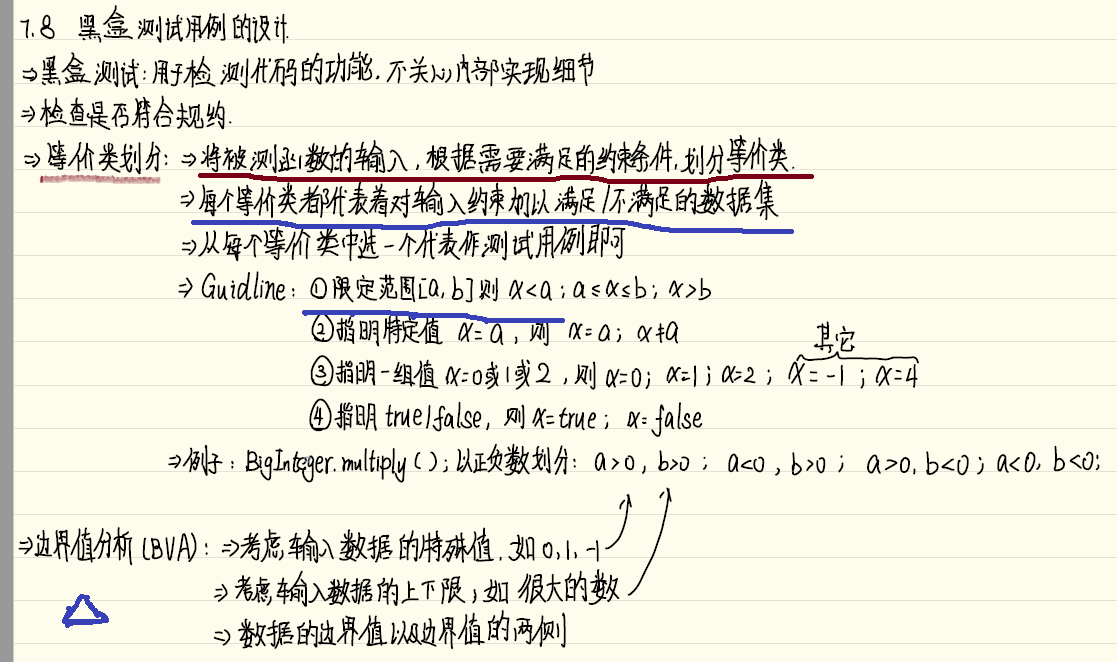
断言一般在开发阶段帮助我们发现bug 实际运行时一般不再使用

不要在断言中操作数据 执行有副作用的代码

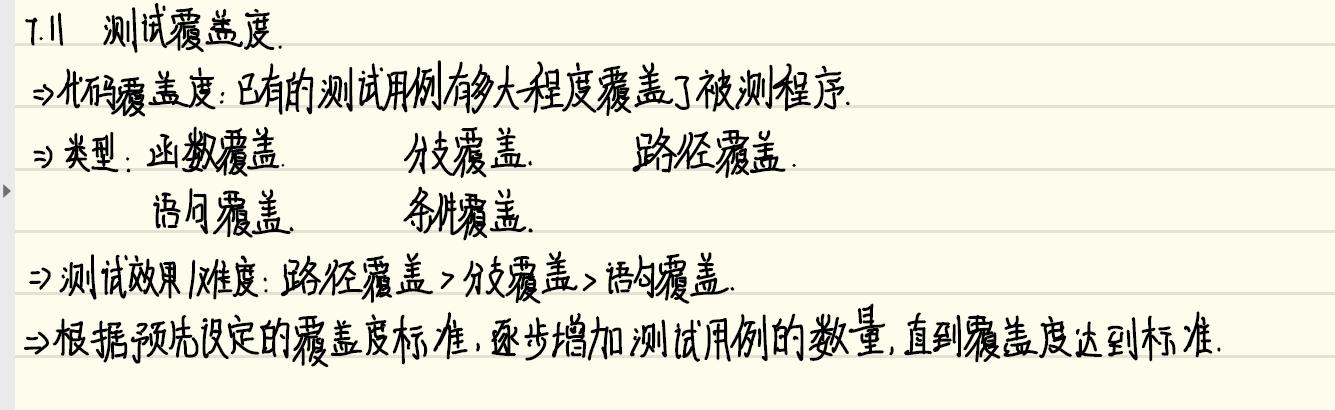
对于pre/post codition 若方法参数来自外部》异常处理

来自自己的其他代码 使用断言

调试 调试的基本过程 重现-定位-修复-反思 常用方法 假设检验法



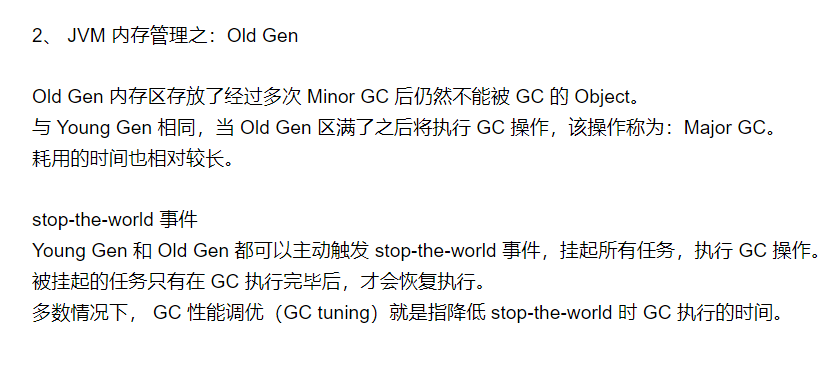


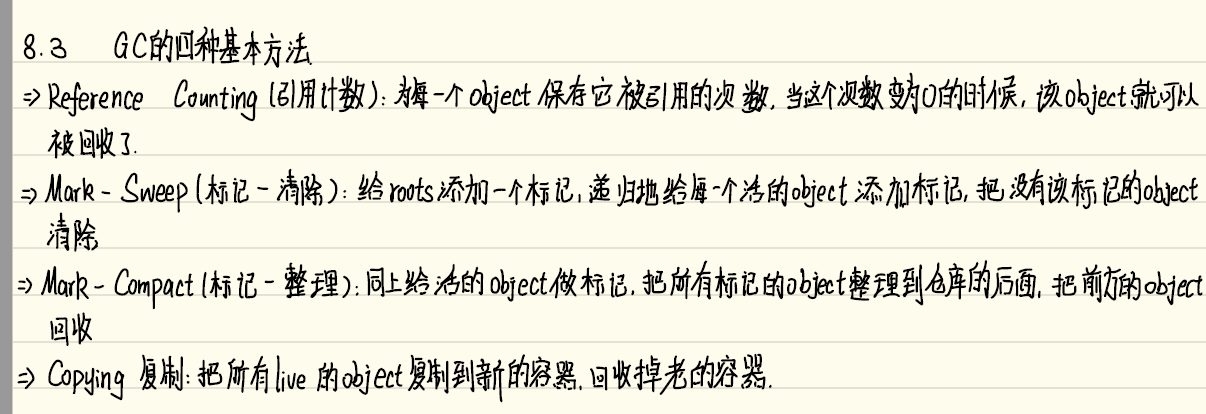


内存管理

三种内存管理模式 1.静态 2.基于栈的动态 3.基于堆的动态







第一个方式 循环引用清不掉

多线程

线程安全的四种策略

Confinement

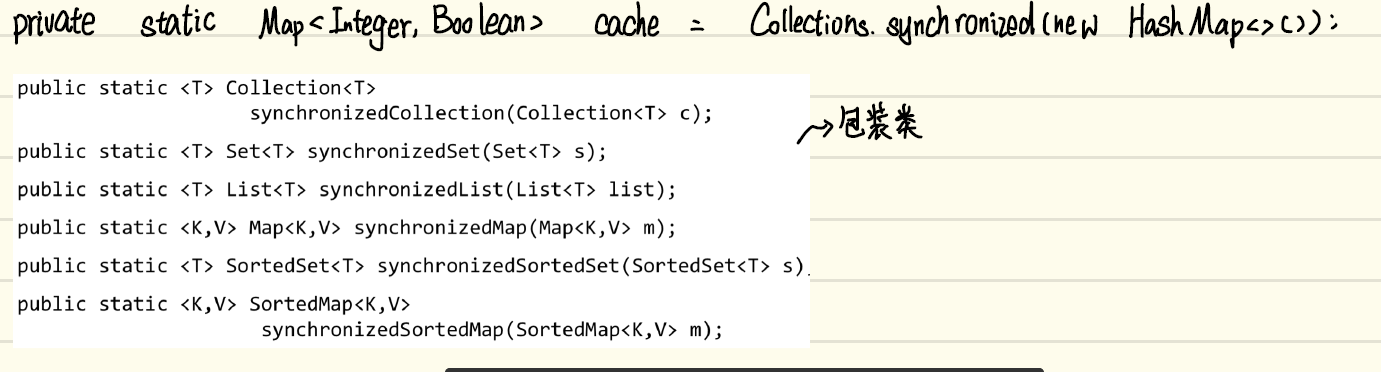
将可变数据限制在某个线程内部，避免竞争 不允许任何线程读写该数据

Immutability

不可变数据类型一般都是thread safe的

Synchronized

如果必须使用mutable类型的数据类型共享数据 java提供了decorator模式



Locks and synchronized机制

程序员自己负责多线程之间对于 mutable数据的共享通过同步策略 避免多个线程同时访问数据