面向对象设计基础

软件构造期末复习

ywy_c_asm

目录

理解方法规约及其条件的含义，会为方法设计规约（必考）
理解ADT的含义及表示独立性，能够分析AF/RI/rep（必考）
理解Java中类、接口、继承、重写、重载、泛型的语言层面上的机制，能够起码看懂其代码，能够分析它们（选择题必考）
能够分析并解决类设计中潜在的表示泄露的风险（必考）
理解ADT对象的等价性，能够分析equals和hashCode的作用或行为（选择题必考）

例子-简单字母集合

- 今天大部分内容我们都将围绕“设计一个表示字母的简单无序集合”的例子展开。
- 这个集合的基本操作如下:
- `new()` 创建一个新的集合
- `add(x)` 将x加入集合，返回加入x后的集合大小
- `find(x)` 检查集合里是否有x
- `ins(S)` 计算当前集合与S的交集

1.ADT

5.类 WordList 有以下四个方法，仅根据其方法定义来确定其类型，最有可能分别是_____

```
public WordList(List<String> words)   creator
public void unique()                  mutator
public WordList getCaptitalized()     producer
public Map<String, Integer> getFrequencies()   observer
```

注：Creator (C), Mutator (M), Producer (P), Observer (O)

A C / M / P / O　　B C / O / M / O

C P / O / C / P　　D C / M / O / O

注意非静态方法(构造函数除外)的输入参数还会隐含一个当前对象！
例如：public static WordList empty(); 这个静态方法就是一个creator，输入参数里没有该类型对象

- 所谓ADT-抽象数据类型，强调作用于数据之上的**操作**，并不关心数据具体是怎么存储的。例如我们的字母集合，只对它定义了操作，它的元素到底是怎么存的？用户不需要知道。可以是字符串，可以是字符数组，可以是哈希表，可以是红黑树……
- ADT可以有4种操作：
- ①**构造器**creator，输入一些其它类型的对象，创建一个该ADT对象。例如创建一个新集合new()或者现实中的构造函数。 **creator** : t* → T
- ②**生产器**producer，通过该ADT的旧对象，创建一个该ADT的新对象，例如计算当前集合与S的交集的方法ins(S)。 **producer** : T+, t* → T
- ③**观察器**observer，通过该ADT本身的数据以及传入参数，计算得到其它类型的值。例如检查集合里是否有x的方法find(x)。 **observer** : T+, t* → t
- ④**变值器**mutator，作出“修改ADT内部数据”的行为，是可变对象与不可变对象的本质区别！例如将x加入集合并返回加入x后的集合大小的方法add(x)。 **mutator** : T+, t* → void | t | T

变值器不一定返回void!

B ADT 某个方法的返回值类型若不是 void，那么它不会是 mutator 方法

2.类与接口

通过new调用具体的类的构造函数，创建具体类的实例对象
通过接口声明变量，让它指向这个具体类的实例对象
不管具体实例是什么类型的，客户端都可以通过接口调用统一的操作!
相当于，接口将具体类实现的ADT操作开放给了客户端。

C　Java 不允许多重继承，故第 5 行 implements 之后不能同时出现 A 和 E

- 接口相当于规定了ADT所需的未实现的操作（方法），这是用户所关注的。
- 类真正地在代码层面实现了接口规定的ADT操作，并且实现了ADT内部的数据存储。

```
interface CharSet{
    public int add(char x);
    public CharSet ins(CharSet s);
    public boolean find(char x);
}
```

不加默认为public

静态检查错误!
接口没给你提供这个方法!

```
//基于String的字符集合
class StringSet implements CharSet{
    private String rep; //类的属性字段
    public StringSet(){
        rep=new String();
    }
    @Override public int add(char x){
        //基于String的集合add的实现
    }
    @Override public CharSet ins(CharSet s){
        //基于String的集合ins的实现
    }
    @Override public boolean find(char x){
        //基于String的集合find的实现
    }
}
```

类的构造函数,相当于ADT操作new()

@Override可加可不加，编译器会自动得知这个方法是接口定义的实现

```
//基于char[]的字符集合
class ArraySet implements CharSet{
    private char[] rep; //类的属性字段
    public ArraySet(){
        rep=new char[0];
    }
    @Override public int add(char x){
        //基于char[]的集合add的实现
    }
    @Override public CharSet ins(CharSet s){
        //基于char[]的集合ins的实现
    }
    @Override public boolean find(char x){
        //基于char[]的集合find的实现
    }
    public char[] getCharArray(){
        //自己的方法
    }
}
```

可同时实现多个接口

这个方法无法通过接口调用

客户端代码：

真正的对象实例

```
CharSet set=new ArraySet();
set.add('a');
char[] array1=set.getCharArray();
char[] array2=((ArraySet)set).getCharArray();
set=new StringSet();
set.add('a');
char[] array3=((ArraySet)set).getCharArray();
```

通过接口调用实例类型中实现的方法

运行时可以转换，set本来就指向ArraySet的对象实例

动态检查错误！set在运行时指向StringSet对象实例，无法转换为ArraySet!

2.类与接口

- 接口实际上可以通过default或者static直接编写方法实现。
- **例1.**我不希望客户端得知具体类名，但客户端需要创建实例对象。**基于static的工厂方法**
- **例2.**我突然想为字母集合ADT增加一个获取集合大小的方法size()，但又不希望修改已经实现的具体类。**在接口中就已实现默认操作的default方法**
- 客户端代码：

```
CharSet set=CharSet.emptySet();
int size=set.size();
```

客户端不需要知道具体类名，尽管这是一个StringSet的实例对象

```
interface CharSet{
    public void add(char x);
    public CharSet ins(CharSet s);
    public boolean find(char x);
    public static CharSet emptySet(){
        //工厂方法，返回一个满足CharSet接口的对象实例
        return new StringSet();
    }
    public default int size(){  //默认方法
        int res=0;
        for(char ch='a';ch<='z';ch++)
            if(this.find(ch))
                res++;
        for(char ch='A';ch<='Z';ch++)
            if(this.find(ch))
                res++;
        return res;
    }
}
```

当然，具体类也可以重新实现size方法

8 以下关于接口的说法，不正确的是____

A 接口中的方法不能用private关键词来修饰

B 接口中的方法不能用final关键词来修饰

C 接口中的方法不能用static关键词来修饰

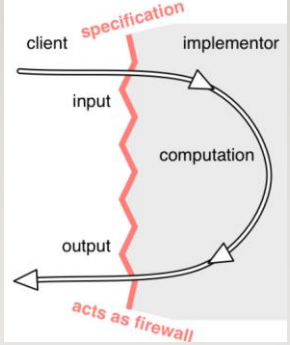
D 接口定义中不能出现implements关键词

接口中的方法原则上是不应该private的，毕竟接口要把类定义的方法公开给客户端，但Java9开始可以，不过必须在接口中给出具体实现

final方法用于类的继承，用在接口里也没有意义啊

一个接口可以通过extends继承另一个甚至多个接口的方法，但显然不能implements

3.方法规约(spec)



规约是客户端与实现者之间签订的“契约”，客户端的输入应当满足**前置条件**，实现者编写的程序应当给出满足**后置条件**的结果。
规约描述了方法的功能以及接口（“能做什么”），不需要依赖（也不应该透露）方法的具体实现。

给人看的注释，需要人为检查是否满足

```
/**
 * 将字母x加入集合
 * @param x 要加入集合的字符，必须是大写或小写英文字母
 * @return 操作后的集合大小
 */
public int add(char x);
```

给编译器看的方法声明，能够对方法接口的调用进行静态检查

ADT中的方法规约在接口CharSet中就应该声明，不应该跟具体实现有关

对于这个add方法，如果客户端故意输入一个不是字母的x，违反前置条件，那么add方法也就不需要满足后置条件，理论上干什么都行

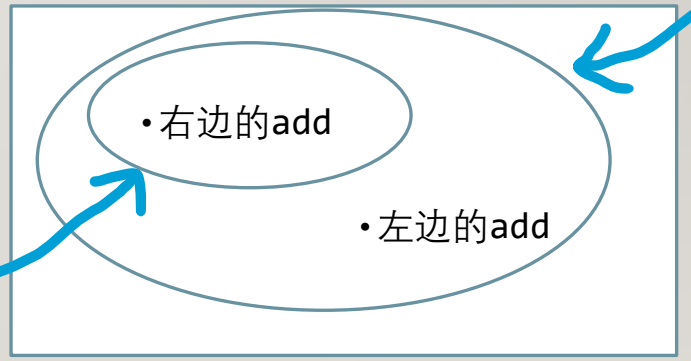
```
/**
 * 将字母x加入集合
 * @param x 要加入集合的字符
 *          若x为大写或小写英文字母则加入集合
 *          否则什么都不做
 * @return 操作后的集合大小
 */
public int add(char x);
```

前置条件（参数要求）

后置条件（返回值要求，或者throws抛出的异常）

对于外圈，客户端必须传入字母

对于内圈，客户端可以传入任意字符



规约图：更强的规约表示为更小的区域（因为满足它的方法更少）

客户端更容易满足　　甚至还能达到更好的效果

更强的规约：**前置条件更弱，后置条件更强**，满足更强规约的方法一定能替代满足更弱规约的功能，客户端肯定更喜欢强规约方法（有更大的自由度），但这增加了实现者的压力……
（例：右边的add的规约强于左边的）

3.方法规约(spec)

(5) 以下关于方法 spec 的说法，不恰当的是___

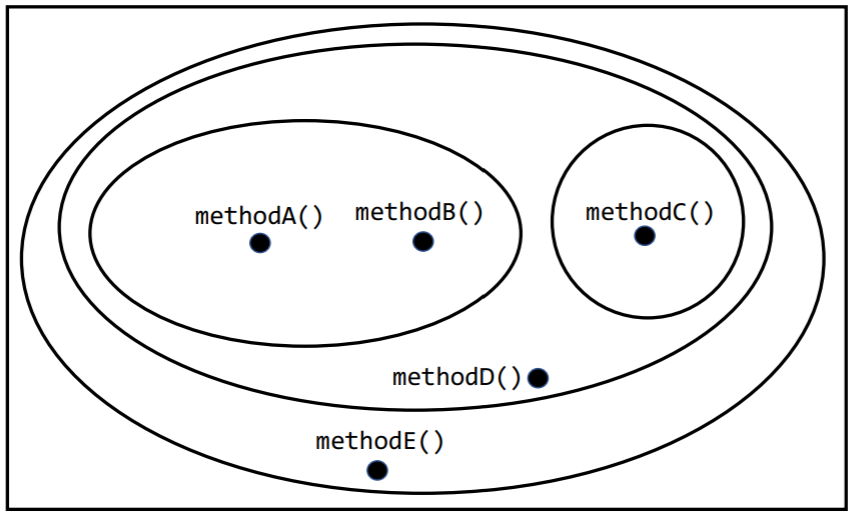
A 若客户端传递进来的参数不满足前置条件，则方法可直接退出或随意返回一个结果

B 方法的 spec 描述里不能使用内部代码中的局部变量或该方法所在类的 private 属性

C 方法 A 的前置条件比方法 B 的前置条件更强，后置条件相同，那么开发者实现 A 的难度要高于实现 B 的难度

D 如果修改了某个方法的 spec 使之强度变弱了，那么 client 调用该方法的代价可能变大了，即 client 需要对调用时传入该方法的参数做更多的检查

8. 阅读下图的 specification diagram，其中黑点代表一个具体的方法实现，以下说法不正确的是



A methodA()和 methodB()的 spec 对客户端程序员来说没有区别

B methodC()的 spec 的强度比 methodD()的 spec 的强度要大

C 若遵循 methodD()的 spec，客户端程序员可以无区别的使用 methodA()和 methodC()

D 在任何可以使用 methodD()的场合，都可以使用 methodE()而不会造成副作用

```
/*A mutable ADT*/
public interface Poll {
    /**
     * 初始化一次投票活动
     * @param candidates  一组候选人，candidates.size()>=1；每个元素代表一个候选人的姓名，
     *                    姓名由字母构成，首位大写，其内部不含空格
     * @param maxSupportNum 每个投票人可投支持票的最大数目，>=1、<=candidates.size()
     * @return 一个新的“投票活动”对象
     */
    static Poll initialize(Set<String> candidates, int maxSupportNum) {
        return new ConcretePoll(candidates, maxSupportNum);
    }

    /**
     * 读取一个投票人的投票，如果选票中不存在四种非法情况，将投票结果加入到记录中，否则抛出异常
     * @param voter 投票人的名字，无限定条件
     * @param votes Key代表候选人名字，Value代表对该候选人的投票结果
     * @throws NoEnoughCandidatesException 如果votes没有覆盖本次投票活动中的所有候选人
     * @throws InvalidCandidatesException 如果votes包含了不在本次投票活动中的候选人
     * @throws InvalidScoresException 如果votes.values()中包含了-1、0、1之外的值
     * @throws BeyondMaxSupportNumException 如果votes中值为1的数目超过了允许支持的最大人数
     */
    void vote(String voter, Map<String, Integer> votes)
            throws NoEnoughCandidatesException, InvalidCandidatesException,
                   InvalidScoresException, BeyondMaxSupportNumException;
```

三（5 分）如果把 vote(...)方法的 spec 中第三个@throws 去掉，在@param votes 那一行的末尾增加“-1 表示反对，0 表示弃权，1 表示支持，其他值不合法”。那么，修改后的 vote(...)的 spec 的强度，相比起修改前发生了什么变化？为什么？

前置条件更弱了，后置条件更强了（抛出的异常更少了），规约更强了，客户端自由度更高，方法更难实现

4.AF/RI/rep

客户端可以看到ADT的操作方法及规约，可以看到抽象空间以及抽象值，但就是**不能看到rep、RI以及AF**，它们涉及到客户端不应该知道的与内部实现有关的细节

```
class Fraction{ //表示分数的ADT
    private int a,b;    //rep
    // RI:  b!=0
    // AF:  AF(a,b)=a/b
    ...
```

这个例子中多个rep可以映射到同样的抽象值

一个便于理解但不太准确的例子，对于黑盒函数 $f(x)=\ln x$，rep就是x，AF就是表达式$\ln x$，表示空间和抽象空间都是实数域，RI为x>0的限制

C　AF 作为一种关系，具备的性质是：满射、双射，但并非总是单射

- 对于ADT，客户端是无法看到其内部表示属性(rep)的，客户端只会看到ADT在表面上展现出来的东西，即，ADT做了一个由表示空间(R)到抽象空间(A)的映射AF。

D　两个 ADT 具有相同 rep 和相同的 RI，那么在客户端程序员眼里它们是等价的

- 不是所有表示属性rep都能映射到相应的抽象值的（即有一些rep是非法的），那么任何时刻ADT的rep都必须满足一定规则(合法)，即表示不变量RI。
- 有时应该引入一个方法checkRep检查rep是否满足RI，像这样：

C　如果 ADT 的任意 constructor 所构造出的 object 都满足 RI、每个 mutator 方法执行结束后都保持 RI 为真，那么该 ADT 的 RI 就永远保持为真

这是一个仅实现者使用的private内部方法，客户端不需要知道

```
private void checkRep(){
    //每次作出修改rep的操作时原则上都应该检查RI
    for(int i=0;i<rep.length();i++){
        char ch=rep.charAt(i);
        assert (ch>='a'&&ch<='z')||(ch>='A'&&ch<='Z');
    }
}
```

```
//基于String的字符集合
class StringSet implements CharSet{
    private String rep; //类的属性字段
    // RI:
    //   rep中仅含有英文字母
    // AF:
    //   AF(rep)={rep[i]|0<=i<rep.length()}
    ...
```

应该以注释形式写出AF和RI

给客户端展现出的是一个字母集合(**抽象值**)，然而实际上的数据是一个字符串(**表示值**)，ADT正是使用**操作**们对这个字符串做了特殊的“解读”，实现了AF

其它可能的RI和AF：

要想满足这个RI可能还要修改add的规约

```
// RI:  rep中仅含有英文字母，并且rep中字符不重复
// AF:  AF(rep)={rep[i]|0<=i<rep.length()}
```

```
// RI:  rep中仅含有英文字母
// AF:  AF(rep)=rep中字符构成的多重集
```

要想完善这个AF可能还要增加“统计某字符出现次数”的方法

4.AF/RI/rep

- 注意！对于不可变类型，它的抽象值一定是不能变的，但表示值可以改变（反正客户端也不知道），前提是改变后通过AF仍然映射到相同的抽象值！

```java
class Fraction{ //分数ADT
    private int a,b; //rep
    //RI:   b!=0
    //AF:   AF(a,b)=a/b
    ...
    @Override public String toString(){
        int g=gcd(a,b); //最大公约数
        a/=g; b/=g; //约分，不改变抽象值，改变rep
        return Integer.toString(a)+"/"+b;
    }
}
```

这丝毫不影响该类
是一个不可变类型

A　Immutable 类的对象，其 rep 自对象创建之后就不能再发生变化

D　一个 immutable 的 ADT，其 rep 可以是 mutable 的

4.AF/RI/rep

```
/*A mutable ADT*/
public interface Poll {
    /**
     * 初始化一次投票活动
     * @param candidates  一组候选人，candidates.size()>=1；每个元素代表一个候选人的姓名，
     *                    姓名由字母构成，首位大写，其内部不含空格
     * @param maxSupportNum 每个投票人可投支持票的最大数目，>=1、<=candidates.size()
     * @return 一个新的“投票活动”对象
     */
    static Poll initialize(Set<String> candidates, int maxSupportNum) {
        return new ConcretePoll(candidates, maxSupportNum);
    }

    /**
     * 读取一个投票人的投票，如果选票中不存在四种非法情况，将投票结果加入到记录中，否则抛出异常
     * @param voter 投票人的名字，无限定条件
     * @param votes Key代表候选人名字，Value代表对该候选人的投票结果
     * @throws NoEnoughCandidatesException 如果votes没有覆盖本次投票活动中的所有候选人
     * @throws InvalidCandidatesException 如果votes包含了不在本次投票活动中的候选人
     * @throws InvalidScoresException 如果votes.values()中包含了-1、0、1之外的值
     * @throws BeyondMaxSupportNumException 如果votes中值为1的数目超过了允许支持的最大人数
     */
    void vote(String voter, Map<String, Integer> votes)
            throws NoEnoughCandidatesException, InvalidCandidatesException,
                   InvalidScoresException, BeyondMaxSupportNumException;
```

五 （10 分）类 ConcretePoll 实现了 Poll 接口。它的 rep 如下：

```
private Set<String> candidates = new HashSet<>(); // 一组候选人
private Set<Vote> votes = new HashSet<>(); // 投票记录
private int maxSupportNum; //每个投票人投票时允许的最大支持数
```

Vote 是一个 immutable ADT，表示投票人对一个候选人的投票，其 rep 为：

```
private String voter;  //投票人名字
private String candidate; //候选人名字
private int score; //投票人给候选人的分数，只能为-1、0、1 之一，
                   //分别表示反对、弃权、支持
```

基于上述 rep 和上页代码，分别写出 ConcretePoll 和 Vote 的 RI。

写RI=写rep满足的规则=“阅读理解”

六 （5 分）针对第五题给出的 ConcretePoll 的 rep，实现了它的 creator 方法，并增加了两个 observer 方法，分别返回某个候选人的所有得票结果、本次投票的所有候选人：

```
public ConcretePoll(Set<String> candidates, int maxSupportNum) {...}
public List<Integer> getVotesByCandidate(String candidate) {...}
public Set<String> listAllCandidates() {...}
```

不考虑它们内部具体如何实现，判断 ConcretePoll 是否可能存在表示泄露。如果可能存在，考虑在上述方法的具体实现代码中采取什么措施进行规避，以注释形式写出 ConcretePoll 的 safety from rep exposure。

5.表示独立性与表示泄露

实现者应该以注释的形式声明如何避免表示泄露：

```
//基于char[]的字符集合
class ArraySet implements CharSet{
    private char[] rep; //类的属性字段
    //Safety from rep explosure:
    //  所有字段都是private的 这是最基本最起码的保护方式
    //  getCharArray返回的是rep的拷贝
```

- 表示独立性是指，客户端使用ADT时无需考虑（也不应该知道，更不应该直接访问到）其内部如何实现，ADT内部表示的变化不应影响外部spec和客户端。
- 如果ADT不幸地让客户端得到了自己内部表示(可变对象)的引用，那么客户端就可以不通过ADT的操作，而可以通过非法后门修改ADT的内部表示，产生表示泄露。

A 一个 mutable 的 ADT，因为其 rep 值可变，存在表示泄露也不会造成危害

```
//基于char[]的字符集合
class ArraySet implements CharSet{
    private char[] rep; //类的属性字段
    ...
    //以数组方式获取字符集合内容
    public char[] getCharArray(){
        return rep;
    }
}
```

产生表示泄露！char[]是mutable的！“主动泄密”

```
ArraySet set=new ArraySet();
set.add('b');
char[] arr=set.getCharArray();
arr[0]='a';
```

客户端通过后门篡改了ArraySet的rep

```
//基于char[]的字符集合
class ArraySet implements CharSet{
    private char[] rep; //类的属性字段
    ...
    public char[] getCharArray(){
        return Arrays.copyOf(rep, rep.length);
    }
}
```

应该像这样做防御性拷贝！返回一个新对象的引用

```
class MyClass{
    private List<Integer> rep;
    public MyClass(List<Integer> list){
        this.rep=list;
    }
    ...
}
```

另外一种典型的“内鬼式”表示泄露，直接把传入的可变对象引用赋给rep

```
List<Integer> list=new ArrayList<>();
MyClass me=new MyClass(list);
list.add(233333);
```

客户端通过传入的内鬼直接在外面改变rep

```
class MyClass{
    private List<Integer> rep;
    public MyClass(List<Integer> list){
        this.rep=new ArrayList<>(list);
    }
}
```

传入时也应该像这样做防御式拷贝

6.继承与重写

对于一个表面类型为父类的引用变量a，通过a.b()调用方法时，在运行时将会动态地检查a是否实际指向一个子类型对象，子类型是否重写了b()方法，如果是那么则调用子类型的重写方法。这一机制实现了子类型多态，即同样的a.b()，可能调用不同的方法。

- 一个类child可以继承另一个类father的全部方法与属性，在运行时存在继承关系的类型对象可以互相转换。子类型也可以重写(override)父类的方法，替换为自己的实现代码。
- 例：我希望对类StringSet进行扩展，创建一个可记录操作日志的类StringLogSet

extends指定父类，Java只允许一个父类

```
class StringLogSet extends StringSet{
    private List<String> log;
    public StringLogSet(){
        super();
        log=new ArrayList<>();
    }
    @Override public int add(char x){
        log.add("Try add a char "+x);
        return super.add(x);
    }
    @Override public boolean find(char x){
        log.add("Try find a char "+x);
        return super.find(x);
    }
    public final List<String> getLog(){
        return new ArrayList<>(log);
    }
}
```

使用super指代自己的父类，这里是先调用父类的构造函数

重写add方法

@Override可加可不加，在代码中注明这是重写方法

参数列表和父类的一致

调用父类定义的add方法

子类自己新增的方法

getLog被声明为final，StringLogSet的子类不能再重写它！

客户端代码：

set尽管表明上是一个StringSet类型的引用变量，但它实际上指向一个StringLogSet对象

```
StringSet set=new StringLogSet();
set.add('a');
set.ins(new StringSet());
set.getLog();
StringLogSet logset=(StringLogSet)set;
List<String> list=logset.getLog();
```

在运行时发现set实际指向StringLogSet对象，那么就调用StringLogSet重写的add方法

ins没被重写，调用StringSet的

静态检查错误！set的表面类型StringSet没getLog方法！

现在可以调用logSet

在运行时尝试向下转换到子类型，若set实际指向的对象与StringLogSet没有继承关系，则转换失败，引发异常

6.继承与重写-“我是什么类型”

s的表面类型是StringSet，这个编译器在静态检查的时候知道，但s具体指向的对象到底是什么类型，编译器并不能知道

- 考虑两个函数func1和func2:

```
static int func1(StringSet s){
    if(s instanceof StringLogSet)
        return 233;
    else
        return 666;
}
```

运行时instanceof将尝试将s转换为StringLogSet，若转换失败则得到false的结果

```
static int func2(StringSet s){
    if(s instanceof StringSet)
        return 233;
    else
        return 666;
}
```

一定要明白，变量在代码中指定的的表面类型，和**运行时**变量实际指向的对象类型，可能是具有继承关系的不同类型

- 分别调用它们:
- func1(new StringSet())=666
- func1(new StringLogSet())=233
- func2(new StringSet())=233
- func2(new StringLogSet())=233

6. 以下代码段中，返回结果为 true 的是____

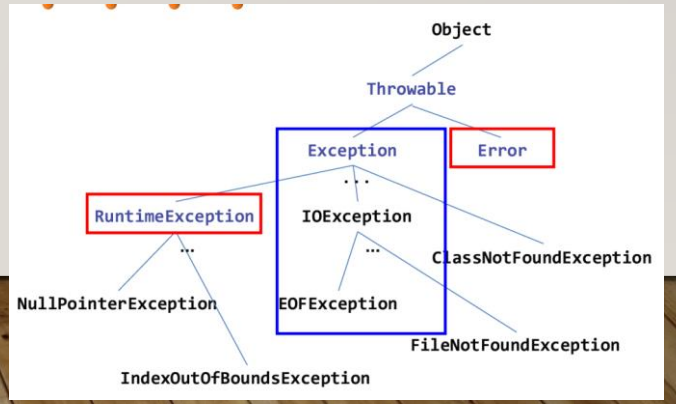
D

```
Number a = new Double(3.0);
Number b = new Integer(0);
a = b;
return a instanceof Double;
```

不能用instanceof判断一个对象是否是真正的父类对象！

总结：Java中常用类的接口实现与继承

- 容器类可以有不同的底层实现，但它们的操作都较为统一，表示为接口实现的形式。例如，ArrayList<T>和LinkedList<T>都实现了接口List<T>，都能实现列表的统一操作，但分别基于数组和链表实现。HashMap<K,V>和TreeMap<K,V>都实现了接口Map<K,V>，都能实现映射的统一操作，但分别基于哈希表和红黑树实现。HashSet<T>和TreeSet<T>都实现了接口Set<T>，都能实现集合的统一操作，但分别基于哈希表和红黑树实现。
- 所有引用类型的最终祖先类都是Object，都具有等价性判断equals()、哈希函数hashCode()和字符串转换toString()方法，可以重写它们。
- Integer、Double等表示数值的引用类型(跟int、double这些非引用的内置类型并非一回事！)都继承自Number类，都具有转换为其它数值类型的方法(例如intValue())。
- 所有异常类都继承自Exception，对于父异常的catch能够捕获子异常。
- **这些Java相关的知识考试的时候可能会默认你知道。**



```
public boolean equals(StringSet s){
    //do sth.
}
```

注意：这玩意并没有在重写Object.equals，参数列表都不一样，这是重载的同名方法！

```
@Override public boolean equals(Object obj){
    if(obj instanceof StringSet){
        StringSet s=(StringSet)obj;
        for(char ch:this.rep.toCharArray())
            if(!s.find(ch)) //枚举this每个字符，判断是否属于s
                return false;
        for(char ch:s.rep.toCharArray())
            if(!this.find(ch)) //枚举s每个字符，判断是否属于this
                return false;
        return true;
    }else return false;
}
```

由于equals的参数必须是Object，所以需要先检查类型是否匹配

7.ADT的等价性与equals()

- ADT的等价性是对于客户端角度而言的，要么，两个对象通过AF映射到相同的抽象值，要么，两个对象能做出效果相同的行为。不一定非得让rep完全一致。

B Override equals()的时候，equals()的代码中需要逐个比较 rep 中每一个域的值是否相等
C 某 ADT 的对象 a 和 b，若 a.equals(b)为假，那么该 ADT 不应存在任何方法 op 使得 a.op()=b.op()

- 所有对象都继承了Object.equals()，我们可以在类中重写它，从而定义自己的等价规则。注意equals()与==不是一回事，后者仅仅判断两个引用是否指向同一对象。如果不重写Object.equals()，那么默认效果和==是一样的。
- 对于可变对象，除了上述的“观察等价性”，还会有一种“行为等价性”，如果两个对象这个时刻等价，那么不管之后干了什么，这两个可变对象仍然是等价的。（一般和==一样，当且仅当是同一对象，**例如StringBuilder,**我们常用的除它之外的类一般都是观察等价性，如String、List）

6. 以下代码段中，返回结果为 true 的是____

A
```
List<String> lst1 = new ArrayList<>();
List<String> lst2 = new LinkedList<>();
lst1.add("a");
lst2.add(new String("a"));
return lst2.equals(lst1);
```

B
```
String s1 = new String("abc");
List<String> list = new ArrayList<String>();
list.add(s1);
s1 = s1.concat("d");
return list.get(0).equals("abcd");
```

C
```
StringBuilder sb1 = new StringBuilder("ABC");
StringBuilder sb2 = new StringBuilder("ABC");
return sb1.equals(sb2);
```

```
StringBuilder sb1=new StringBuilder("haha");
StringBuilder sb2=new StringBuilder("haha");
StringBuilder sb3=sb1;
String s1=new String("HIT");
String s2=new String("HIT");
List<Integer> l1=new ArrayList<>();
List<Integer> l2=new LinkedList<>();
l1.add(233);
l2.add(233);
```

s1==s2	false
sb1==sb2	false
sb1==sb3	true
s1.equals(s2)	true
sb1.equals(sb2)	false
sb1.equals(sb3)	true
l1.equals(l2)	true

8.hashCode

10. 以下关于 hashCode()的说法，最恰当的是___
A 如果不需要把 ADT 对象放入集合类中使用，那么该 ADT 无需重写 hashCode()
B 如果两个 ADT 对象不等价，那么它们的 hashCode()返回结果不应该相等
C 如果父类已重写了 hashCode()、子类的 rep 中没有引入新的属性且没有重写 equals(..)，那么子类中无需重写 hashCode()
D 在 hashCode()内部生成 hash code 时，要考虑 ADT 的 rep 中包含的所有属性

- Object.hashCode()计算对象的哈希值，将对象映射为一个整数，使得等价的对象具有相同的哈希值。好的哈希算法应当使得不等价对象的哈希值尽量不同。

C hashCode()中如果只包含“return 31;”语句，在功能实现上也是没有问题的

A Immutable 的 ADT 对象 a 和 b，若 a.hashCode()=b.hashCode()，那么 a.equals(b)一定为真

- hashCode()一般用在基于哈希表的集合类中(HashMap、HashSet)。查找一个对象时，先通过hashCode直接找到对象对应的桶，再在桶里一个个用equals比较。
- Object.hashCode()默认直接返回对象的地址，即对象哈希值相同当且仅当为同一个对象(引用一样)，当使用了自定义的等价规则时，需要重写hashCode。

StringSet.hashCode的一种可能的实现
{a,b,d}->$(1011)_2$=11

```
@Override public int hashCode(){
    long hash=0;
    for(int i=0;i<26;i++)
        if(this.find((char)(i+'a')))
            hash|=(1l<<i);
    for(int i=0;i<26;i++)
        if(this.find((char)(i+'A')))
            hash|=(1l<<(i+26));
    return (int)(hash%998244353);
}
```

hashCode的实现一般可以通过进制数的方式，将不同的组合部分表示为进制数中的数位，最终得到一个大数，一般可通过取模使其离散。
（考试应该不会考这些，了解即可）

9.重载

7. 以下关于 overload 的说法，正确的是___
A overload 是 OOP 中一种典型的参数化多态机制
B 两个 overload 的方法，要有不同的参数数目
C 两个 overload 的方法，返回值类型要遵循“协变”原则
D 父类型 A 和子类型 B 中可以存在 overload 的方法

overload实际上属于特设多态(ad-hoc)，这种多态下，同一操作针对不同类型表现出不同的行为

- 重载(overload)机制使得同一个类中的多个方法可以有相同的名字，前提是它们有长度不同的参数列表，或者对应不同的参数类型，起码，得让编译器在进行**静态检查**的时候通过你调用时传入的参数判断实际上应该选择哪个方法。重载也可以发生在父类与子类之间(子类重载父类的方法)。

```java
class MyClass{
    private int rep;
    public MyClass(int rep){
        this.rep=rep;
    }
    public void func(int a,int b){
        System.out.println("func1");
    }
    public int func(double a,double b){
        System.out.println("func2");
        return 0;
    }
    public boolean equals(MyClass m){
        return this.rep==m.rep;
    }
}
```

重载函数的返回值可以没有任何关系

重载父类的equals，而不是重写！

```java
MyClass a=new MyClass(1);
a.func(1,1); func1
a.func(1.0,1.0); func2
MyClass b=new MyClass(1);
Object c=b;
System.out.println(a.equals(b)); true(MyClass.equals)
System.out.println(a.equals(c)); false(Object.equals)
```

编译器在编译时选择合适的方法调用

（但如果两个重载方法的参数列表十分接近，编译器决策不了到底应该调用哪个，静态检查错误）

10.泛型

3种基本的多态:
①特设多态(同一操作，不同类型不同行为)，重载
②参数化多态(操作与类型无关)，泛型
③子类型多态(同一对象可能属于多种类型)，继承/重写

```
public interface A extends B {
    public static A m() {...;}
    List<Object> n(Number a);
}
public class C extends D implements A,E {
    @Override
    public List<String> n(Number d) {
        ...
    }
```

D 类 C 中的方法 n(..)是对接口 A 中方法 n(..)的合法 override，符合 LSP 原则

- 泛型是**参数化多态**的实现机制，它能够将类型作为类/方法的参数，使得操作与类型无关。
- 例：我们发现为CharSet这个ADT设计的操作实际上可以扩充到任意类型的集合，因此可以将其使用泛型扩展为Set<T>以及具体类ConcreteSet<T>。

```
interface Set<T>{ 带有类型参数T的接口
    public int add(T x);
    public boolean find(T x);
    public Set<T> ins(Set<T> s);
}
class ConcreteSet<T> implements Set<T>{
    private List<T> rep;
    public ConcreteSet(){
        rep=new ArrayList<>();
    }
    @Override public int add(T x) {...}
    @Override public boolean find(T x) {...}
    @Override public Set<T> ins(Set<T> s) {...}
}
```

泛型不存在于运行时！编译器会为它们自动生成两个真正的类ConcreteSet_Integer和ConcreteSet_Number，并将类定义中的T分别直接替换为Integer和Number（类型擦除）

```
ConcreteSet<Integer> s1=new ConcreteSet<Integer>();
ConcreteSet<Number> s2=new ConcreteSet<Number>();
s1.add(1);
s2.add(1);
s2=s1;
```

1是Integer，Integer继承自Number

静态检查错误！ConcreteSet<Integer>和ConcreteSet<Number>没有继承关系！（泛型的不可协变性）

加extends表示T必须是Number的后代类，同理若加super表示T必须是xxx的祖先类

```
class NumberSet<T extends Number> implements Set<T>{
    ...
    @Override public int add(T x) {
        int val=x.intValue();
        ... 此时已经告诉编译器限定了T是Number的后代类，那么T的变量就可以当作Number变量使用
    }
```

9 针对方法void m(Set<? super Integer> set)，以下___作为参数传递进去不是对它的合法调用（注：Number是Integer的父类型）

A HashSet<Object> set　　B Set<Integer> set

C Set<? extends Number> set　　D TreeSet<Number> set

万一这个参数实际指向的是一个TreeSet<Double>……

```
void func(List<? extends Number> list){
    这个带extends的通配符限定了传入参数必须是类型参数T为Number子类的List<T>，例如List<Integer>可以，List<Object>不可以，同理也可以用super声明
}
```

谢谢大家

- qwq