哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

5 Designing Specification
设计规约

Wang Zhongjie
rainy@hit.edu.cn

May 24, 2021

Objective of this lecture

- **Understand preconditions and postconditions in method specifications, and be able to write correct specifications (方法的规约)**
- **What preconditions and postconditions are, and what they mean for the implementor and the client of a method (前置/后置条件)**
- **Understand underdetermined specs, and be able to identify and assess nondeterminism (欠定规约、非确定规约)**
- **Understand declarative vs. operational specs, and be able to write declarative specs (陈述式、操作式规约)**
- **Understand strength of preconditions, postconditions, and specs; and be able to compare spec strength (规约的强度及其比较)**
- **Be able to write coherent, useful specifications of appropriate strength (如何写出好的规约)**

Outline

1. **Function / method in programming language**
2. **Specification: Programming for communication**
   - Why specification is needed
   - Behavioral equivalence
   - Specification structure: pre-condition and post-condition
   - Testing and verifying specifications
3. **Designing specifications**
   - Classifying specifications
   - Diagramming specifications
   - Quality of a specification
4. **Summary**

上一节关注了编程语言中的“数据类型”、“变量”、“值”，尤其是mutable和immutable的类型/值/引用

本节转向“方法/函数/操作”如何定义——编程中的“动词”、规约

Reading

- **MIT 6.031：06、07**
- **CMU 17-214：Sep 05**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

1 Functions & methods in programming languages

Method

```
public static void threeLines() {
        STATEMENTS;
}

public static void main(String[] arguments){
        System.out.println("Line 1");
        threeLines();
        System.out.println("Line 2");
}
```

Parameters

```
[…] NAME (TYPE NAME, TYPE NAME) {
        STATEMENTS
}

To call:
        NAME(arg1, arg2);
```

- **Attention**: parameter type mismatch when calling a method – static checking 参数类型是否匹配，在静态类型检查阶段完成

Return Values

```
public static TYPE NAME() {
        STATEMENTS;
        return EXPRESSION;
}
```

void means “no type”

返回值类型是否匹配，也在静态类型检查阶段完成

Variable Scope 变量的作用域

Immutable

```
class SquareChange {
    public static void printSquare(int x){
        System.out.println("printSquare x = " + x);
        x = x * x;
        System.out.println("printSquare x = " + x);
    }
    public static void main(String[] arguments){
        int x = 5;
        System.out.println("main x = " + x);
        printSquare(x);
        System.out.println("main x = " + x);
    }
}
```

5

25

5

5

Methods: Building Blocks

- **Big programs are built out of small methods**
- **Methods can be individually developed, tested and reused**
- **User of method does not need to know how it works --- this is called "abstraction"**

"方法"是程序的"积木"，可以被独立开发、测试、复用

使用"方法"的客户端，无需了解方法内部具体如何工作—"抽象"

A complete method

3n+1问题（Hailstone问题）

3n+1问题是一个简单有趣而又没有解决的数学问题。这个问题是由Collatz在1937年提出的。克拉兹问题（Collatz problem）也被叫做hailstone问题、3n+1问题、Hasse算法问题、Kakutani算法问题、Thwaites猜想或者Ulam问题。

问题如下：

（1）输入一个正整数n；

（2）如果n=1则结束；

（3）如果n是奇数，则n变为3n+1，否则n变为n/2；

（4）转入第（2）步。

克拉兹问题的特殊之处在于：尽管很容易将这个问题讲清楚，但直到今天仍不能保证这个问题的算法对所有可能的输入都有效——即至今没有人证明对所有的正整数该过程都终止。

方法的实现体implementation

```
    }
}
```

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2 Specification: Programming for communication

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(1) Documenting in programming

Java API documentation: an example

java.util

Class LinkedList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSequentialList<E>
        java.util.LinkedList<E>

Type Parameters:

`E` - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Note that this implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new LinkedList(...));
```

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`List`, `ArrayList`, Serialized Form

Java API documentation: an example

Field Summary

Fields inherited from class java.util.Al

modCount

Constructor Summary

Constructors

Constructor and Description
LinkedList() Constructs an empty list.
LinkedList(Collection<? extends E> c) Constructs a list containing the elements of the specified

Method Summary

Methods

Modifier and Type	Method and Description
boolean	add(E e) Appends the specified element t
void	add(int index, E element) Inserts the specified element at t
boolean	addAll(Collection<? extends] Appends all of the elements in th returned by the specified collecti
boolean	addAll(int index, Collectio Inserts all of the elements in the
void	addFirst(E e) Inserts the specified element at t
void	addLast(E e) Appends the specified element t
void	clear() Removes all of the elements fror
Object	clone() Returns a shallow copy of this Li
boolean	contains(Object o) Returns true if this list contains t

Constructor Detail

LinkedList

public LinkedList()

Constructs an empty list.

LinkedList

public LinkedList(Collection<? extends E> c)

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Parameters:

c - the collection whose elements are to be placed into this list

Throws:

NullPointerException - if the specified collection is null

indexOf

public int indexOf(Object o)

Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. More formally, returns the lowest index i such that (o==null ? get(i)==null : o.equals(get(i))), or -1 if there is no such index.

Specified by:

indexOf in interface List<E>

Overrides:

indexOf in class AbstractList<E>

Parameters:

o - element to search for

Returns:

the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element

Java API documentation: an example

- **Class hierarchy and a list of implemented interfaces.**
- **Direct subclasses, and for an interface, implementing classes.**
- **A description of the class**
- **Constructor summary**
- **Method summary lists all the methods we can call**
- **Detailed descriptions of each method and constructor**
  - The **method signature**: we see the return type, the method name, and the parameters. We also see *exceptions*. For now, those usually mean errors the method can run into.
  - The full **description**.
  - **Parameters**: descriptions of the method arguments.
  - And a description of what the method **returns**.

Documenting Assumptions

- **Writing the type of a variable down documents an assumption about it: e.g., this variable will always refer to an integer.** 变量的数据类型定义
  - Java actually checks this assumption at compile time, and guarantees that there's no place in your program where you violated this assumption.
- **Declaring a variable `final` is also a form of documentation, a claim that the variable will never change after its initial assignment.** `final`关键字定义了设计决策："不可改变"
  - Java checks that too, statically.
- **How about the assumptions of functions / methods?**
- 代码本身就蕴含着你的设计决策，但是远远不够

Programming for communication

- **Why do we need to write down our assumptions?**
  - Because programming is full of them, and if we don't write them down, we won't remember them, and other people who need to read or change our programs later won't know them. They'll have to guess.
  - 为什么要写出“假设”？第一：自己记不住；第二：别人不懂。

- **Programs have to be written with two goals in mind:**
  - Communicating with the computer. First persuading the compiler that your program is sensible – syntactically correct and type-correct. Then getting the logic right so that it gives the right results at runtime. 代码中蕴含的“设计决策”：给编译器读
  - Communicating with other people. Making the program easy to understand, so that when somebody has to fix it, improve it, or adapt it in the future, they can do so. 注释形式的“设计决策”：给自己和别人读

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(2) Specification and Contract (of a method)

Specifications (or called Contract)

- **Specifications are the linchpin 关键 of teamwork. It's impossible to delegate responsibility for implementing a method without a specification.** 没规约，没法写程序；即使写出来，也不知道对错
- **The specification acts as a contract 契约: the implementer is responsible for meeting the contract, and a client that uses the method can rely on the contract.** 程序与客户端之间达成的一致
  - States method's and caller's responsibilities
  - Defines what it means for implementation to be correct
- **Specifications place demands on both parties: when the specification has a precondition, the client has responsibilities too. Spec**给“供需双方”都确定了责任，在调用的时候双方都要遵守
  - If you pay me this amount on this schedule…
  - I will build a with the following detailed specification
  - Some contracts have remedies for nonperformance

Why specifications?

- **Reality:**
  - Many of the nastiest bugs in programs arise because of misunderstandings about behavior at the interface between two pieces of code. 很多bug来自于双方之间的误解
  - Although every programmer has specifications in mind, not all programmers write them down. As a result, different programmers on a team have *different* specifications in mind. 不写下来，那么不同开发者的理解就可能不同
  - When the program fails, it's hard to determine where the error is. 没有规约，难以定位错误
- **Advantages:**
  - Precise specifications in the code let you apportion blame to code fragments, and can spare you the agony of puzzling over where a fix should go. 精确的规约，有助于区分责任
  - Specifications are good for the client of a method because they spare the task of reading code. 客户端无需阅读调用函数的代码，只需理解spec即可

An example of specification

- **A method `add()` of a Java class `BigInteger`**

Specification from the API documentation :

add

`public BigInteger add(BigInteger val)`

Returns a BigInteger whose value is `(this + val)`.

Parameters:

`val` - value to be added to this BigInteger.

Returns:

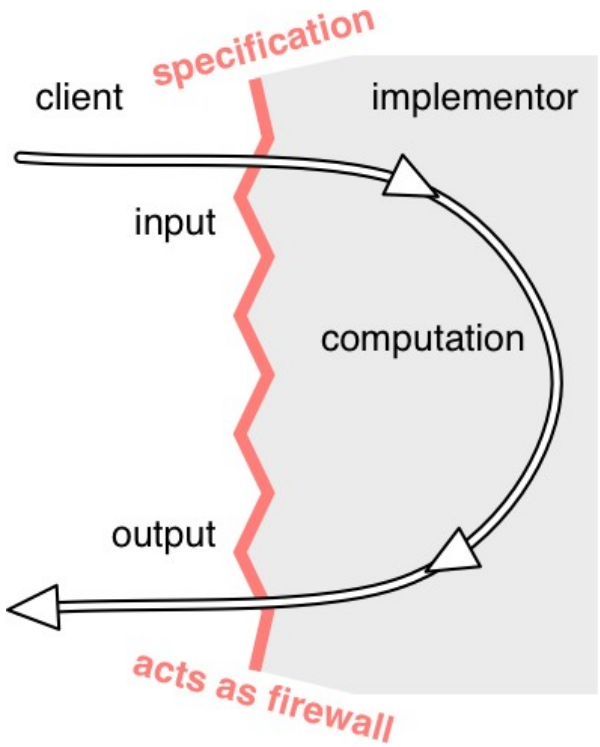
`this + val`

Method body from Java 8 source :

```
if (val.signum == 0)
    return this;
if (signum == 0)
    return val;
if (val.signum == signum)
    return new BigInteger(add(mag, val.mag), signum);

int cmp = compareMagnitude(val);
if (cmp == 0)
    return ZERO;
int[] resultMag = (cmp > 0 ? subtract(mag, val.mag)
                   : subtract(val.mag, mag));
resultMag = trustedStripLeadingZeroInts(resultMag);

return new BigInteger(resultMag, cmp == signum ? 1 : -1);
```

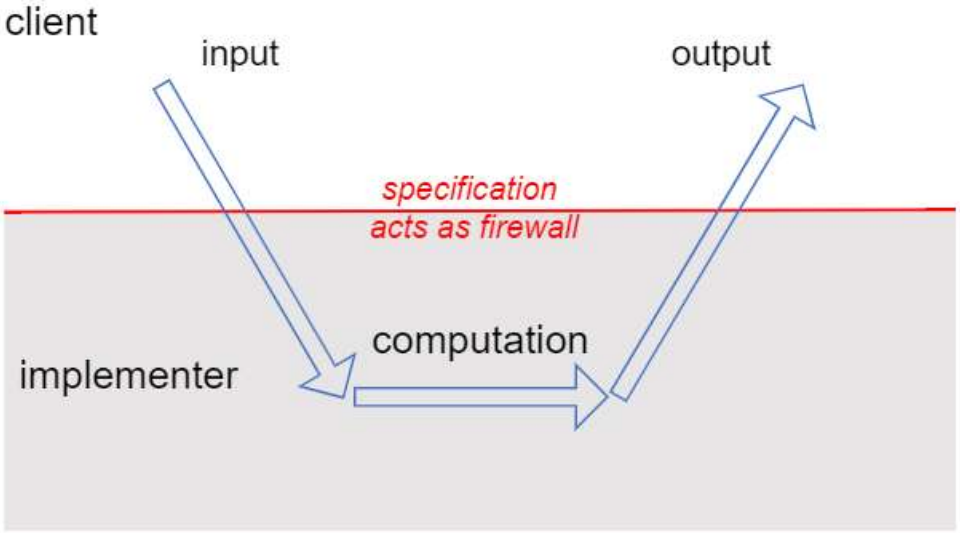
Specification (contract)

- Specifications are good for the implementer of a method because they give the implementor **freedom to change** the implementation without telling clients. 规约可以隔离“变化”，无需通知客户端
- Specifications can **make code faster**, too. 规约也可以提高代码效率
- **The contract acts as a *firewall* between client and implementor.** 规约：扮演“防火墙”角色
  - It shields the client from the details of the *workings* of the unit.
  - It shields the implementor from the details of the *usage* of the unit.
  - This firewall results in *decoupling*, allowing the code of the unit and the code of a client to be changed independently, so long as the changes respect the specification.
  - 解耦，不需了解具体实现



Specification (contract)

- **Agreement between an object and its user**
  - Method signature (type specifications) 输入/输出的数据类型
  - Functionality and correctness expectations 功能和正确性
  - Performance expectations 性能
- **What the method does, not how it does it** 只讲“能做什么”，不讲“怎么实现”
  - Interface (API), not implementation



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(3) Behavioral equivalence

Behavioral equivalence (行为等价性)

- To determine **behavioral equivalence** , the question is whether we could substitute one implementation for the other (是否可相互替换).

```
static int findFirst(int[] arr, int val) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == val) return i;
    }
    return arr.length;
}

static int findLast(int[] arr, int val) {
    for (int i = arr.length -1 ; i >= 0; i--) {
        if (arr[i] == val) return i;
    }
    return -1;
}
```

这两个函数是否等价？

行为不同，但对用户来说“是否等价”？

When `val` is missing, `findFirst` returns the length of `arr` and `findLast` returns `-1`;

When `val` appears twice, `findFirst` returns the lower index and `findLast` returns the higher.

- The notion of equivalence is **in the eye of the client** (站在客户端视角看行为等价性).

Behavioral equivalence (行为等价性)

- To determine **behavioral equivalence** , the question is whether we could substitute one implementation for the other (是否可相互替换).

```
static int findFirst(int[] arr, int val) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == val) return i;
    }
    return arr.length;
}

static int findLast(int[] arr, int val) {
    for (int i = arr.length -1 ; i >= 0; i--) {
        if (arr[i] == val) return i;
    }
    return -1;
}
```

这两个函数是否等价？

行为不同，但对用户来说“是否等价”？

But when `val` occurs at exactly one index of the array, the two methods behave the same.

Whenever they call the method, they will be passing in an `arr` with exactly one element `val`.

For such clients, these two methods are the same.

- The notion of equivalence is **in the eye of the client** (站在客户端视角看行为等价性).

Behavioral equivalence

- In order to make it possible to substitute one implementation for another, and to know when this is acceptable, we need a specification that states exactly what the client depends on 根据规约判断是否行为等价

```
static int find(int[] arr, int val)
  requires: val occurs exactly once in arr
  effects:  returns index i such that arr[i] = val
```

这两个函数符合这个规约，故它们等价。

Classroom Exercises

```
static int findFirst(int[] a, int val) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == val) return i;
    }
    return a.length;
}
```

Suppose clients only care about calling the `find` method when they know `val` occurs exactly once in `a`.

In this case, are `findFirst` and `findLast` behaviorally equivalent?

```
static int findLast(int[] a, int val) {
    for (int i = a.length - 1 ; i >= 0; i--) {
        if (a[i] == val) return i;
    }
    return -1;
}
```

Suppose clients only care that the `find` method should return:

- any index `i` such that `a[i] == val`, if `val` is in `a`
- any integer `j` such that `j` is not a valid array index, otherwise

In this case, are `findFirst` and `findLast` behaviorally equivalent?

Short summary

单纯的看实现代码，并不足以判定不同的implmentation是否是“行为等价的”

需要根据代码的spec（开发者与client之间形成的contract）判定行为等价性

在编写代码之前，需要弄清楚spec如何协商形成、如何撰写

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

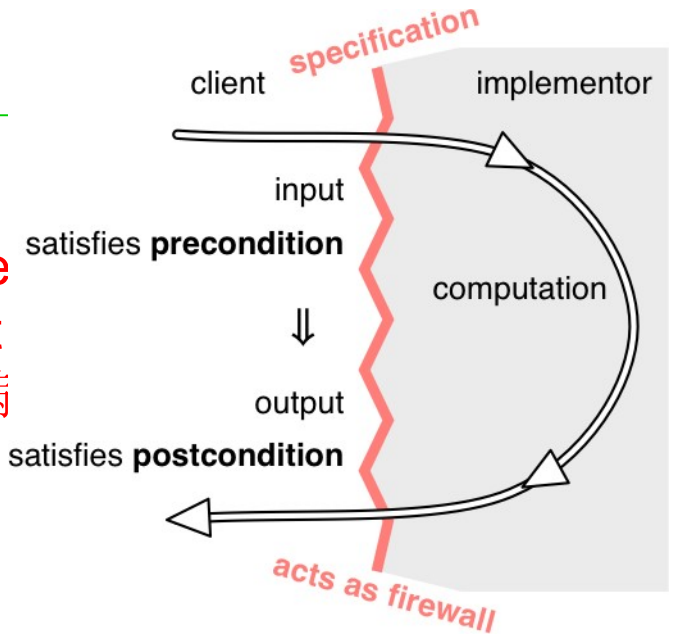
(4) Specification structure: pre-condition and post-condition

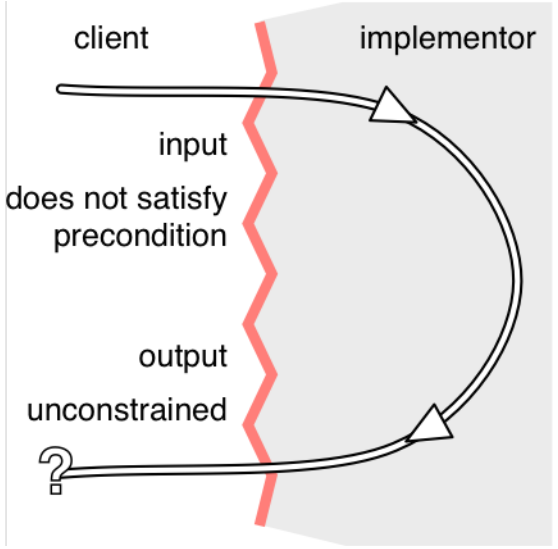
Specification Structure

- **A specification of a method consists of several clauses:**
  - ***Precondition*** , indicated by the keyword *requires*
  - ***Postcondition*** , indicated by the keyword *effects*
  - **Exceptional behavior**: what it does if precondition violated 意外行为
- The **precondition** is an obligation on the client (i.e., the caller of the method). It's a condition over the state in which the method is invoked. 前置条件：对客户端的约束，在使用方法时必须满足的条件
- The **postcondition** is an obligation on the implementer of the method. 后置条件：对开发者的约束，方法结束时必须满足的条件
- If the precondition holds for the invoking state, the method is obliged to obey the postcondition, by returning appropriate values, throwing specified exceptions, modifying or not modifying objects, and so on. 契约：如果前置条件满足了，后置条件必须满足

Specification Structure

- The overall structure is a logical implication: *if* the precondition holds when the method is called, *then* the postcondition must hold when the method completes. 前置条件满足，则后置条件必须满足。
- If the precondition does *not* hold when the method is called, the implementation is *not* bound by the postcondition.
  - It is free to do anything, including not terminating, throwing an exception, returning arbitrary results, making arbitrary modifications, etc.
  - 前置条件不满足，则方法可做任何事情。
  - “你违约在先，我自然不遵守承诺”





Classroom Exercises

```
static int find(int[] arr, int val)
```

requires: `val` occurs exactly once in `arr`

effects: returns index `i` such that `arr[i]` = `val`

As the implementer of `find`, which are legal?

- if `arr` is empty, return `0`
- if `arr` is empty, throw an `exception`
- if `val` occurs twice in `arr`, throw an `exception`
- if `val` occurs twice in `arr`, set all the values in `arr` to `0`, then throw an `exception`
- if `arr` is not empty but `val` doesn't occur, pick an `index` at random, set it to `val`, and return that `index`
- if `arr[0]` is `val`, continue checking the rest of the array and return the highest `index` where you find `val` (or `0` if you don't find it again)

The client has violated the precondition. As implementers, we are free to do ***anything***.

If `arr[0]` is the only occurrence of `val`? If we do find another `val` in the array?

Classroom Exercises

```
static int find(int[] arr, int val)
```

requires: `val` occurs exactly once in `arr`

effects: returns index `i` such that `arr[i]` = `val`

- if `arr` is empty, return `0`
- if `arr` is empty, throw an `exception`
- if `val` occurs twice in `arr`, throw an `exception`
- if `val` occurs twice in `arr`, set all the values in `arr` to `0`, then throw an `exception`

As the implementer of `find`, why would you choose to throw an `exception` if precondition is violated?

When our precondition is violated, the client has a bug.
We can make that bug easier to find and fix by failing fast, even though we are not obligated to do so.

Specifications in Java

- Java's **static type declarations** *are* effectively part of the precondition and postcondition of a method, a part that is automatically checked and enforced by the compiler. 静态类型声明是一种规约，可据此进行静态类型检查**static checking**。
- The rest of the contract must be described in a **comment** preceding the method, and generally depends on human beings to check it and guarantee it. 方法前的注释也是一种规约，但需人工判定其是否满足
- Parameters are described by `@param` clauses and results are described by `@return` and `@throws` clauses.
- Put the **preconditions into** `@param` where possible, and **postconditions into** `@return` **and** `@throws`.

Specifications in Java

```
static int find(int[] arr, int val)
  requires: val occurs exactly once in arr
  effects:  returns index i such that arr[i] = val
```

```
/**
 * 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)
```

Recall the code in Lab1

```java
/**
 * Determine inside angles of a regular polygon.
 *
 * There is a simple formula for calculating the inside angles of a polygon;
 * you should derive it and use it here.
 *
 * @param sides number of sides, where sides must be > 2
 * @return angle in degrees, where 0 <= angle < 360
 */
public static double calculateRegularPolygonAngle(int sides) {
    throw new RuntimeException("implement me!");
}

/**
 * Given a sequence of points, calculate the Bearing adjustments needed to get from each point
 * to the next.
 *
 * Assumes that the turtle starts at the first point given, facing up (i.e. 0 degrees).
 * For each subsequent point, assumes that the turtle is still facing in the direction it was
 * facing when it moved to the previous point.
 * You should use calculateBearingToPoint() to implement this function.
 *
 * @param xCoords list of x-coordinates (must be same length as yCoords)
 * @param yCoords list of y-coordinates (must be same length as xCoords)
 * @return list of Bearing adjustments between points, of size 0 if (# of points) == 0,
 *         otherwise of size (# of points) - 1
 */
public static List<Double> calculateBearings(List<Integer> xCoords, List<Integer> yCoords) {
    throw new RuntimeException("implement me!");
}
```

Classroom Exercises

Given this spec:

```
static boolean isPalindrome(String word)
 requires: word contains only alphanumeric characters
   effects: returns true if and only if word is a palindrome
```

Here is a flawed attempt to write the spec in Javadoc:

```
/*
 * Check if a word is a palindrome.
 * A palindrome is a sequence of characters
 * that reads the same forwards and backwards.
 * @param String word
 * @requires word contains only alphanumeric characters
 * @effects returns true if and only if word is a palindrome
 * @return boolean
 */
```

Which lines in the Javadoc comment are problematic?

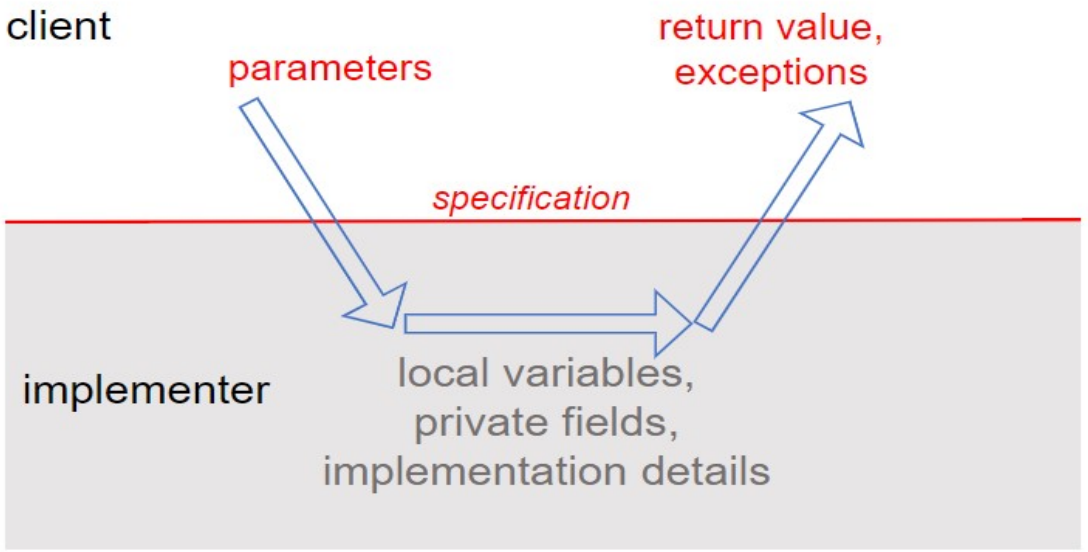
Classroom Exercises

```
/**
 * Calculate the potential energy of a mass in Earth's gravitational field.
 * @param altitude altitude in meters relative to sea level
 * @return potential energy in joules
 */
static double calculateGravitationalPotentialEnergy(double altitude);
```

This exercise shows that in a spec, parts of the precondition and postcondition may be found in places other than the `@param` and `@return` clauses, so it's important to read carefully.

What a specification may talk about

- A specification of a method can talk about the parameters and return value of the method, but it should never talk about local variables of the method or private fields of the method's class. *只讨论方法的参数值和返回值，不讨论方法的类的局部变量或私有域。*
  - You should consider the implementation invisible to the reader of the spec.
  - In Java, the source code of the method is often unavailable to the reader of your spec, because the Javadoc tool extracts the spec comments from your code and renders them as HTML.



Specifications for mutating methods

- Example 1: a mutating method

```
static boolean addAll(List<T> list1, List<T> list2)
  requires: list1 != list2
  effects:  modifies list1 by adding the elements of list2 to the end of
              it, and returns true if list1 changed as a result of call
```

- Example 2: a mutating method

```
static void sort(List<String> lst)
  requires: nothing
  effects:  puts lst in sorted order, i.e. lst[i] <= lst[j]
              for all 0 <= i < j < lst.size()
```

- Example 3: a method that does not mutate its argument

```
static List<String> toLowerCase(List<String> lst)
  requires: nothing
  effects:  returns a new list t where t[i] = lst[i].toLowerCase()
```

Specifications for mutating methods

- **If the *effects* do not explicitly say that an input can be mutated, then we assume mutation of the input is implicitly disallowed.** 除非在后置条件里声明过，否则方法内部不应该改变输入参数
- **Virtually all programmers would assume the same thing. Surprise mutations lead to terrible bugs.** 应尽量遵循此规则，尽量不设计**mutating**的**spec**，否则就容易引发**bugs**。
- **Convention** 程序员之间应达成的默契：除非**spec**必须如此，否则不应修改输入参数
  - Mutation is disallowed unless stated otherwise .
  - No mutation of the inputs

- **Mutable objects can make simple specification/contracts very complex** 尽量避免使用**mutable**的对象
- **Mutable objects reduce changeability**

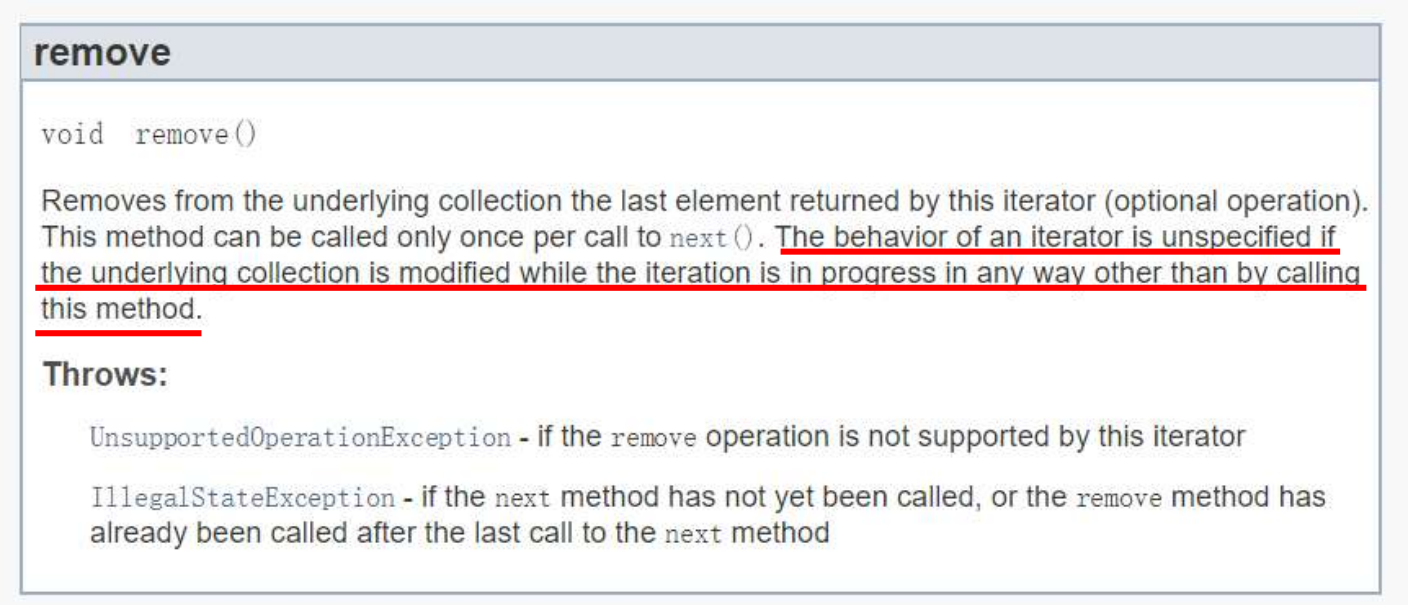
Mutable objects make simple contracts complex

- Multiple references to the same mutable object (**aliases**别名 for the object) may mean that multiple places in your program — possibly widely separated — are relying on that object to remain consistent.
  程序中可能有很多变量指向同一个可变对象（别名）
- To put it in terms of specifications, contracts can't be enforced in just one place anymore, e.g. between the client of a class and the implementer of a class. 无法强迫类的实现体和客户端不保存可变变量的“别名”
- Contracts involving mutable objects now depend on the good behavior of everyone who has a reference to the mutable object.

不能靠程序员的“道德”，要靠严格的“契约”

Mutable objects make simple contracts complex

- **As a symptom 症状 of this non-local contract phenomenon, consider the Java collections classes, which are normally documented with very clear contracts on the client and implementer of a class.**
  - Try to find where it documents the crucial requirement on the client that you can't modify a collection while you're iterating over it.



remove

`void remove()`

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to `next()`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Throws:

`UnsupportedOperationException` - if the `remove` operation is not supported by this iterator

`IllegalStateException` - if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method

Mutable objects make simple contracts complex

- The need to reason about global properties like this make it much harder to understand, and be confident in the correctness of, programs with mutable data structures.

避免使用可变的全局变量！

- We still have to do it — for performance and convenience — but we pay a big cost in bug safety for doing so.

但是为了性能原因，有时候却不得不用。

这对程序的安全性造成了巨大破坏。

Mutable objects reduce changeability

- Mutable objects make the contracts between clients and implementers more complicated, and reduce the freedom of the client and implementer to change.
  - In other words, using *objects* that are allowed to change makes the *code* harder to change. 可变数据类型导致程序修改变得异常困难
- **An example:** a method to looks up a username in database and returns the user's 9-digit identifier

```
/**
 * @param username username of person to look up
 * @return the 9-digit MIT identifier for username.
 * @throws NoSuchUserException if nobody with username is in MIT's database
 */
public static char[] getMitId(String username) throws NoSuchUserException {
    // ... look up username in MIT's database and return the 9-digit ID
}
```

- A client using this method to print out a user's identifier:

```
char[] id = getMitId("bitdiddle");
System.out.println(id);
```

Mutable objects reduce changeability

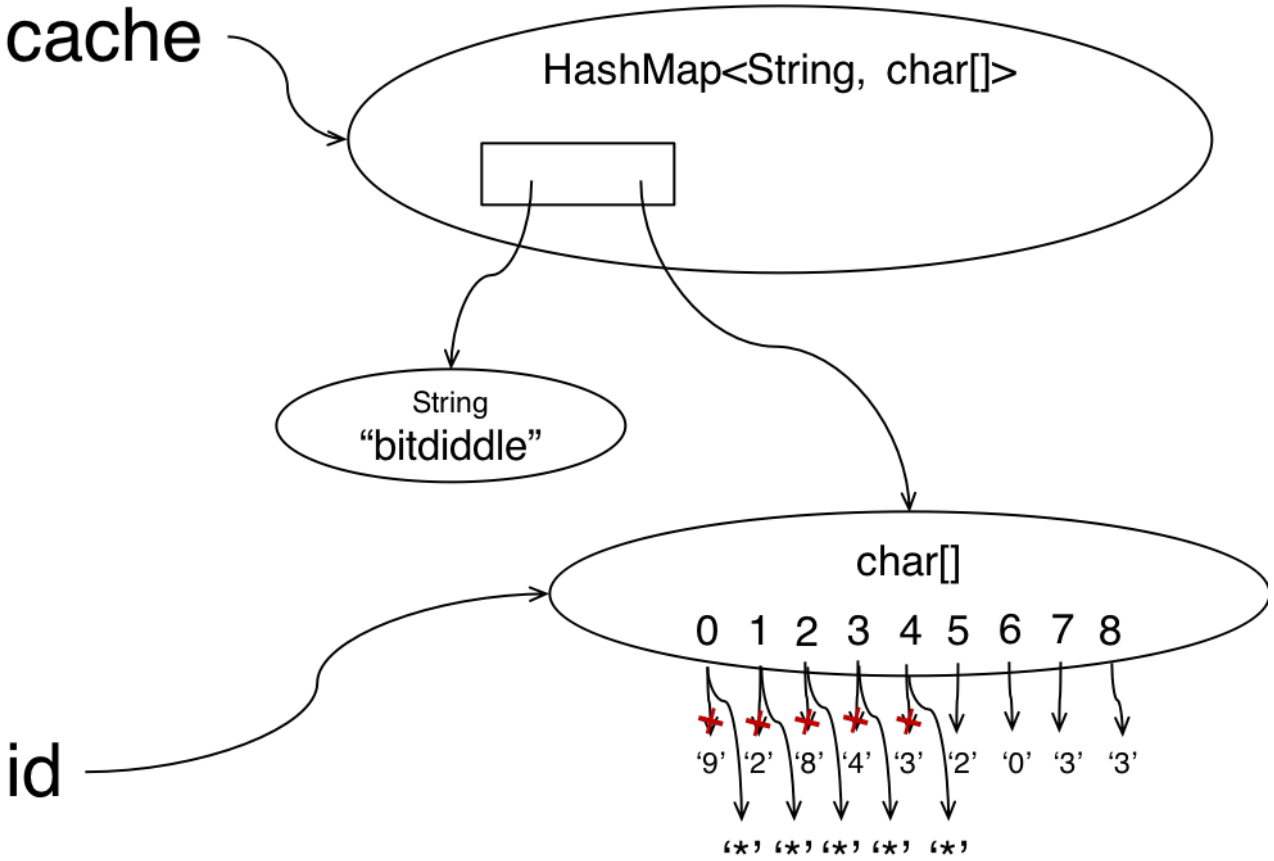
- **Now both the client and the implementor separately decide to make a change.**
  - The client is worried about the user's privacy, and decides to obscure the first 5 digits of the `id`:
  - The implementer is worried about the speed and load on the database, so the implementer introduces a `cache` that remembers usernames that have been looked up:

```
char[] id = getMitId("bitdiddle");
for (int i = 0; i < 5; ++i) {
    id[i] = '*';
}
System.out.println(id);
```

```
private static Map<String, char[]> cache = new HashMap<String, char[]>();

public static char[] getMitId(String username) throws NoSuchUserException {
    // see if it's in the cache already
    if (cache.containsKey(username)) {
        return cache.get(username);
    }

    // ... look up username in MIT's database ...

    // store it in the cache for future lookups
    cache.put(username, id);
    return id;
}
```

- **What will happen?**

Mutable objects reduce changeability



Mutable objects reduce changeability

- **Sharing a mutable object complicates a contract.**
- **Who's to blame here?** 谁为此事负责？
  - Was the client obliged not to modify the object it got back?
  - Was the implementer obliged not to hold on to the object that it returned?

- **A possible way of clarifying the spec:**

```
public static char[] getMitId(String username) throws NoSuchUserException
  requires: nothing
  effects: returns an array containing the 9-digit MIT identifier of username,
           or throws NoSuchUserException if nobody with username is in MIT's
           database. Caller may never modify the returned array.
```

- **How about this spec?**
  - It's a lifetime contract! 完全建立在客户端开发者的“良心”之上，不可靠！

Mutable objects reduce changeability

- **How about this one?**

```
public static char[] getMitId(String username) throws NoSuchUserException
  requires: nothing
  effects: returns a new array containing the 9-digit MIT identifier of username,
           or throws NoSuchUserException if nobody with username is in MIT's
           database.
```

- This spec at least says that the array has to be fresh.
- But does it keep the implementer from holding an alias to that new array?
- Does it keep the implementer from changing that array or reusing it in the future for something else?
- 这回责任又到了开发者这一边的“良心”...都靠不住！

Mutable objects reduce changeability

- How about this one?

```
public static String getMitId(String username) throws NoSuchUserException
  requires: nothing
  effects: returns the 9-digit MIT identifier of username, or throws
           NoSuchUserException if nobody with username is in MIT's database.
```

- The immutable `String` return value provides a *guarantee* that the client and the implementer will never step on each other the way they could with `char` arrays.
- It doesn't depend on a programmer reading the spec comment carefully.
- String is *immutable* . Not only that, but this approach (unlike the previous one) gives the implementer the freedom to introduce a cache — a performance improvement. 关键就在于“不可变”，在规约里限定住

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

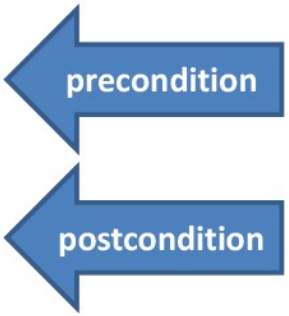
(5)* Testing and verifying specifications

Formal contract specification

http://www.eecs.ucf.edu/~leavens/JML/index.shtml

- **Java Modelling Language (JML)**

```
/*@ requires len >= 0 && array != null && array.length == len;
  @
  @ ensures \result ==
  @            (\sum int j;  0 <= j && j < len;  array[j]);
  @*/
int total(int array[], int len);
```



- **This is a theoretical approach with advantages**
  - Runtime checks generated automatically
  - Basis for formal verification
  - Automatic analysis tools
- **Disadvantages**
  - Requires a lot of work
  - Impractical in the large
  - Some aspects of behavior not amenable to formal specification

Textual specification - Javadoc

- **Practical approach**
- **Documenting** every parameter, return value, every exception (checked and unchecked),what the method does, including Purpose, **s**ide effects, any thread safety issues, any performance issues
- **Do not document implementation details**

```
/**
 * Returns the element at the specified position of this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant time.
 * In some implementations, it may run in time proportional to the
 * element position.
 *
 * @param index position of element to return; must be non-negative and
 *              less than the size of this list.
 * @return the element at the specified position of this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

postcondition

precondition

Semantic correctness adherence to contracts

- **Compiler ensures types are correct (static type checking)**
  - Prevents many runtime errors, such as "`Method Not Found`" and "`Cannot add boolean to int`"
- **Static analysis tools (e.g., FindBugs) recognize many common problems (bug patterns)**
  - For example: Overriding `equals` without overriding `hashCode`
- **But how do you ensure semantic correctness?**

Formal verification

- Use mathematical methods to prove correctness with respect to the formal specification
- Formally prove that all possible executions of an implementation fulfill the specification
- Manual effort; partial automation; not automatically decidable

"Testing shows the presence, not the absence of bugs."

——Edsger W. Dijkstra, 1969

Testing

- **Executing the program with selected inputs in a controlled environment**
- **Goals**
  - Reveal bugs, so they can be fixed (main goal)
  - Assess quality
  - Clarify the specification, documentation

Chapter 6 Robustness

“Beware of bugs in the above code;

I have only proved it correct, not tried it.”

——Donald Knuth, 1977

Black-box testing

- **Black-box testing:** to check if the tested program follow the specified specification in an implementation-independent way.
- Your test cases should not count on any concrete implemented behavior. Test cases must obey the contract, just like every other client.
- **An example specification:**

```
static int find(int[] arr, int val)
  requires: val occurs in arr
  effects:  returns index i such that arr[i] = val
```

- **The test case:**

```
int[] array = new int[] { 7, 7, 7 };
assertEquals(0, find(array, 7));  // bad test case: violates the spec
assertEquals(7, array[find(array, 7)]);  // correct
```

This test case has assumed a specific implementation that `find` always returns the lowest index.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

3 Designing specifications

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(1) Classifying specifications

Comparing specifications

- How **deterministic**不可逆转的 it is. Does the spec define only a single possible output for a given input, or allow the implementor to choose from a set of legal outputs? 规约的确定性
- How **declarative** it is. Does the spec just characterize *what* the output should be, or does it explicitly say *how* to compute the output? 规约的陈述性
- How **strong** it is. Does the spec have a small set of legal implementations, or a large set? 规约的强度

- “What makes some specifications better than others?”
- 用于判断“哪个规约更好”

Stronger vs. weaker specs

- **How to compare the behaviors of two specifications to decide whether it's safe to replace the old spec with the new spec? 如何比较两个规约，以判断是否可以用一个规约替换另一个？**
- **规约的强度S_2>=S_1** A specification S_2 **is stronger than or equal to** a specification S_1 if
  - S_2's precondition is weaker than or equal to S_1's 前置条件更弱
  - S_2's postcondition is stronger than or equal to S_1's, for the states that satisfy S_1's precondition. 后置条件更强

Then an implementation that satisfies S_2 can be used to satisfy S_1 as well, and it's safe to replace S_1 with S_2 in program. **就可以用S_2替代S_1**

spec变强：更放松的前置条件+更严格的后置条件

- **Rules:**
  - *Weaken the precondition*: placing fewer demands on a client will never upset them. *对他人宽容*
  - *Strengthen the post-condition*, which means making more promises. *对自己严格*

Stronger vs. weaker specs

- **Original spec:**

```
static int find_ExactlyOne(int[] a, int val)
  requires: val occurs exactly once in a
  effects:  returns index i such that a[i] = val
```

- **A stronger spec:**

```
static int find_OneOrMore,AnyIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns index i such that a[i] = val
```

- **A much stronger spec:**

```
static int find_OneOrMore,FirstIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns lowest index i such that a[i] = val
```

Stronger vs. weaker specs

- **How about these two?**

```
static int find_ExactlyOne(int[] a, int val)
  requires: val occurs exactly once in a
  effects:  returns index i such that a[i] = val
```

```
static int find_CanBeMissing(int[] a, int val)
  requires: nothing
  effects:  returns index i such that a[i] = val,
              or -1 if no such i
```

前置条件更弱了

在满足$\texttt{find}_{\texttt{ExactlyOne}}$的
前置条件的情况下，
该后置条件无变化

Stronger vs. weaker specs

- **How about these two?**

```
static int find_OneOrMore,AnyIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns index i such that a[i] = val
```

```
static int find_CanBeMissing(int[] a, int val)
  requires: nothing
  effects:  returns index i such that a[i] = val,
              or -1 if no such i
```

前置条件更弱了

在满足$find_{OneOrMore, AnyIndex}$的前置条件的情况下，该后置条件无变化

Stronger vs. weaker specs

- **How about these two?**

```
static int findOneOrMore,FirstIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns lowest index i such that a[i] = val
```

```
static int findCanBeMissing(int[] a, int val)
  requires: nothing
  effects:  returns index i such that a[i] = val,
              or -1 if no such i
```

前置条件更弱了

在满足$find_{OneOrMore, FirstIndex}$的前置条件的情况下，该后置条件弱化了：没有返回lowest index

Stronger vs. weaker specs

- If S_3 is neither stronger nor weaker than S_1, there specs. might overlap *部分重叠* (such that there exist implementations that satisfy only S_1, only S_3, and both S_1 and S_3) or might be disjoint脱节.

- In both cases, S_1 and S_3 are incomparable不可比.

Exercise

- **When a specification is strengthened:**
  - Fewer implementations satisfy it
  - More implementations satisfy it
  - Fewer clients can use it
  - More clients can use it
  - None of the above

A stronger spec might have a weaker precondition and/or stronger postcondition.

In both cases, the implementor must be more careful; but clients with more varied inputs or more specific needs might now be able to make use of the stronger spec.
越强的规约，意味着implementor的自由度和责任越重，而client的责任越轻。

Deterministic vs. underdetermined specs

- **Deterministic**: when presented with a state satisfying the precondition, the outcome is completely determined.
  - Only one return value and one final state is possible.
  - There are no valid inputs for which there is more than one valid output.

```
static int findFirst(int[] arr, int val) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == val) return i;
    }
    return arr.length;
}
```

```
static int findLast(int[] arr, int val) {
    for (int i = arr.length - 1 ; i >= 0; i--) {
        if (arr[i] == val) return i;
    }
    return -1;
}
```

```
static int findExactlyOne(int[] arr, int val)
  requires: val occurs exactly once in arr
  effects:  returns index i such that arr[i] = val
```

确定的规约：给定一个满足precondition的输入，其输出是唯一的、明确的

Deterministic vs. underdetermined specs

- **Under-deterministic:** specification allows multiple valid outputs for the same input. 欠定的规约：同一个输入可以有多个输出

```
static int find_OneOrMore,AnyIndex(int[] arr, int val)
  requires: val occurs in arr
  effects:  returns index i such that arr[i] = val
```

- **Nondeterministic:** sometimes behaves one way and sometimes another, even if called in the same program with the same inputs (e.g., depending on random or timing) 非确定的规约：同一个输入，多次执行时得到的输出可能不同
  - To avoid the confusion, we'll refer to specifications that are not deterministic as underdetermined. 为避免混乱，not d.. == under d..
- **Underdeterminism** in specifications offers a choice that is made by the implementor at implementation time.
  - An underdetermined spec is typically implemented by a fully-deterministic implementation. 欠定的规约通常有确定的实现

Deterministic vs. underdetermined specs

Under-deterministic!

```
static int find_OneOrMore,AnyIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns index i such that a[i] = val
```

Every legal input has exactly one legal output?

```
static int find(int[] arr, int val)
  requires: val occurs exactly once in arr
  effects:  returns index i such that arr[i] = val
```

Deterministic!

```
static int find(int[] arr, int val)
  effects: returns largest index i such that
             arr[i] = val, or -1 if no such i
```

```
static int find(int[] arr, int val)
  requires: val occurs in arr
  effects: returns largest index i such that arr[i] = val
```

Declarative vs. operational specs

- **Operational** specifications give a series of steps that the method performs; pseudocode descriptions are operational. 操作式规约，例如：伪代码
- **Declarative** specifications don't give details of intermediate steps. Instead, they just give properties of the final outcome, and how it's related to the initial state. 声明式规约：没有内部实现的描述，只有“初-终”状态
- **Declarative specifications are preferable.** 声明式规约更有价值
  - They're usually shorter, easier to understand, and most importantly, don't inadvertently expose implementation details that a client may rely on.
- **Why operational spec. exists?**
  - Programmers use the spec to explain the implementation for a maintainer.
  - Don't do that. When it's necessary, use comments within the body of the method, not in the spec comment. 内部实现的细节不在规约里呈现，放在代码实现体内部注释里呈现。

Declarative spec.

- **Standard:** the clearest, for clients and maintainers of the code.

```
static boolean startsWith(String str, String prefix)
 effects: returns true if and only if there exists String suffix
            such that prefix + suffix == str
```

```
static boolean startsWith(String str, String prefix)
 effects: returns true if and only if there exists integer i
            such that str.substring(0, i) == prefix
```

```
static boolean startsWith(String str, String prefix)
 effects: returns true if the first prefix.length() characters of str
            are the characters of prefix, false otherwise
```

Declarative vs. operational specs

```
static String join(String delimiter, String[] elements)
  effects: append together the strings in elements, but at each step,
           if there are more elements left, insert delimiter
```

Which of the following are declarative?

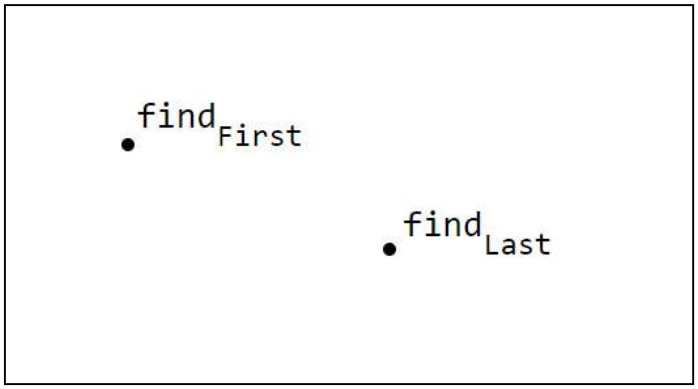
- *effects*: returns the result of adding all elements to a new `StringJoiner(delimiter)`
- *effects*: returns the result of looping through elements and alternately appending an element and the delimiter
- *effects*: returns concatenation of elements in order, with delimiter inserted between each pair of adjacent elements

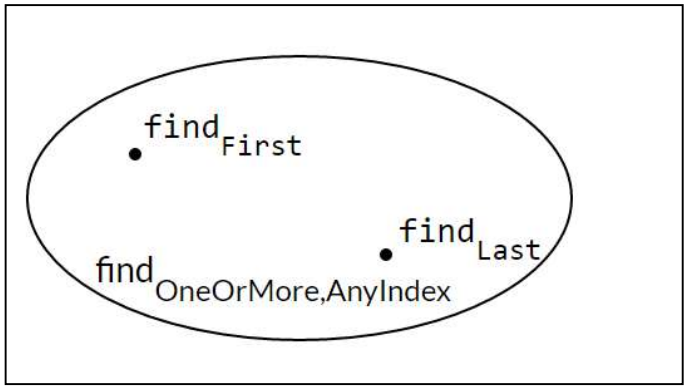
哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(2) Diagramming specifications

Diagramming specifications

- Each point in this space represents a method implementation.

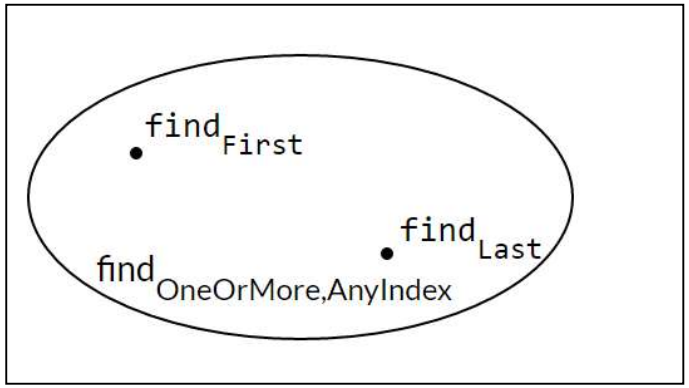




- A specification defines a region in the space of all possible implementations.
- A given implementation either behaves according to the spec, satisfying the precondition-implies-postcondition contract (it is inside the region), or it does not (outside the region). 某个具体实现，若满足规约，则落在其范围内；否则，在其之外。

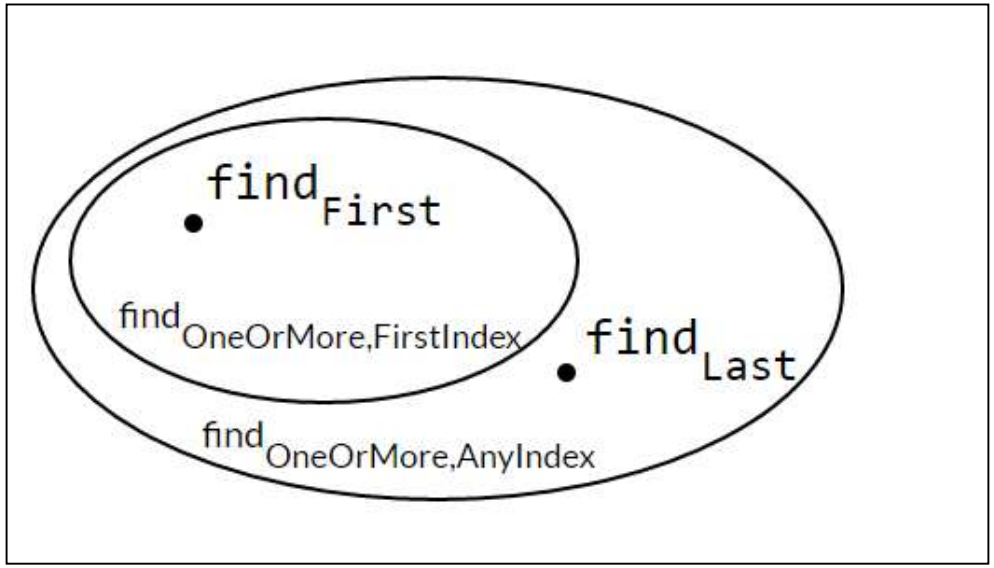
Diagramming specifications

- Implementors have the freedom to move around inside the spec, changing their code without fear of upsetting a client. 程序员可以在规约的范围内自由选择实现方式
  - This is crucial in order for the implementor to be able to improve the performance of their algorithm, the clarity of their code, or to change their approach when they discover a bug, etc.
- Clients don't know which implementation they will get. 客户端无需了解具体使用了哪个实现
  - They must respect the spec, but also have the freedom to change how they're using the implementation without fear that it will suddenly break.



Diagramming specifications

- When S_2 is stronger than S_1, it defines a *smaller* region in this diagram. 更强的规约，表达为更小的区域
- A weaker specification defines a larger region.



Strengthening the postcondition means for implementors: it means they have less freedom, the requirements on their output are stronger 更强的后置条件意味着实现的自由度更低了-➔在图中的面积更小

Weakening the precondition means: implementations will have to handle new inputs that were previously excluded by the spec. 更弱的前置条件意味着实现时要处理更多的可能输入，实现的自由度低了➔面积更小

Diagramming specifications

```
static int find_ExactlyOne(int[] a, int val)
  requires: val occurs exactly once in a
  effects:  returns index i such that a[i] = val
```

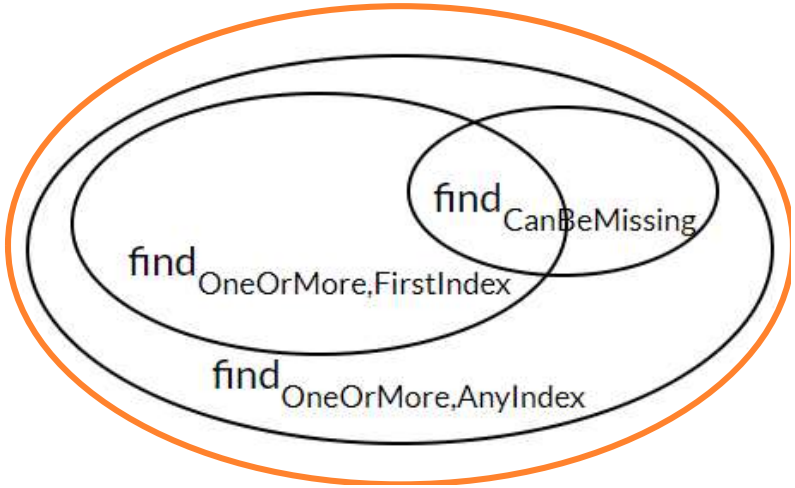
```
static int find_OneOrMore,AnyIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns index i such that a[i] = val
```

```
static int find_OneOrMore,FirstIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns lowest index i such that a[i] = val
```

```
static int find_CanBeMissing(int[] a, int val)
  requires: nothing
  effects:  returns index i such that a[i] = val,
              or -1 if no such i
```

```
ExactlyOne
< OneOrMore, AnyIndex
< OneOrMore, FirstIndex

ExactlyOne
< OneOrMore, AnyIndex
< CanBeMissing

OneOrMore, FirstIndex
CanBeMissing
```
无法比较

Where should `ExactlyOne` be?



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(3) Designing good specifications

Quality of a specification

- What makes a good method? Designing a method means primarily writing a specification.

- About the form of the specification: it should obviously be succinct言简意赅, clear, and well-structured, so that it's EASY TO READ.
- The content of the specification, however, is harder to prescribe. There are no infallible 一贯正确的 rules, but there are some useful guidelines.
- 一个好的“方法”设计，并不是你的代码写的多么好，而是你对该方法的**spec**设计得如何。
  - 一方面：client用着舒服
  - 另一方面：开发者编着舒服

(1) The specification should be coherent (内聚的)

- The spec shouldn't have lots of different cases. Long argument lists, deeply nested if-statements, and boolean flags are all signs of trouble. Spec描述的功能应单一、简单、易理解

```
static int sumFind(int[] a, int[] b, int val)
  effects: returns the sum of all indices in arrays a and b at which
             val appears
```

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

/**
 * Update longestWord to be the longest element of words, and print
 * the number of elements with length > LONG_WORD_LENGTH to the console.
 * @param words list to search for long words
 */
public static void countLongWords(List<String> words)
```

Separating those two responsibilities into two different methods will make them simpler (easy to understand) and more useful in other contexts (ready for change).

- In addition to terrible use of global variables and printing instead of returning, the specification is not coherent — it does two different things, counting words and finding the longest word. 该规约做了两件事，所以要分离开形成两个方法。

(2) The results of a call should be informative

信息丰富的

```
static V put (Map<K,V> map, K key, V val)
  requires: val may be null, and map may contain null values
  effects:  inserts (key, val) into the mapping,
              overriding any existing mapping for key, and
              returns old value for key, unless none,
              in which case it returns null
```

- **If `null` is returned, you can't tell whether the key was not bound previously, or whether it was in fact bound to `null`.**
- **This is not a very good design, because the return value is useless unless you know for sure that you didn't insert `null`.**

不能让客户端产生理解的歧义

(3) The specification should be strong enough

- The spec should give clients a strong enough guarantee in the general case — it needs to satisfy their basic requirements.
  - We must use extra care when specifying the special cases, to make sure they don't undermine what would otherwise be a useful method.
  - For example, there's no point throwing an exception for a bad argument but allowing arbitrary mutations, because a client won't be able to determine what mutations have actually been made.

```
static void addAll(List<T> list1, List<T> list2)
  effects: adds the elements of list2 to list1,
             unless it encounters a null element,
             at which point it throws a NullPointerException
```

没有充分阐明遇到null之后参数是否变化

  - If a `NullPointerException` is thrown, the client is left to figure out on their own which elements of `list2` actually made it to `list1` .

- 太弱的**spec**，**client**不放心、不敢用 **(**因为没有给出足够的承诺**)**。
  开发者应尽可能考虑各种特殊情况，在**post-condition**给出处理措施。

(4) The specification should also be weak enough

```
static File open(String filename)
  effects: opens a file named filename
```

- **This is a bad specification.**
  - It lacks important details: is the file opened for reading or writing? Does it already exist or is it created?
  - It's too strong, since there's no way it can guarantee to open a file. The process in which it runs may lack permission to open a file, or there might be some problem with the file system beyond the control of the program.
- **Instead, the specification should say something much weaker**: it attempts to open a file, and if it succeeds, the file has certain properties.
- 太强的**spec**，在很多特殊情况下难以达到，给开发者增加了实现的难度（**client**当然非常高兴）。

(5) The specification should use *abstract types*

- **Writing our specification with abstract types gives more freedom to both the client and the implementor.** 在规约里使用抽象类型，可以给方法的实现体与客户端更大的自由度
- **In Java, this often means using an interface type, like** `Map` **or** `Reader`**, instead of specific implementation types like** `HashMap` **or** `FileReader`**.**
  - Abstract notions like a `List` or `Set`
  - Particular implementations like `ArrayList` or `HashSet`.

```
static ArrayList<T> reverse(ArrayList<T> list)
  effects: returns a new list which is the reversal of list, i.e.
             newList[i] == list[n-i-1]
             for all 0 <= i < n, where n = list.size()
```

- **This forces the client to pass in an** `ArrayList`**, and forces the implementor to return an** `ArrayList`**, even if there might be alternative** `List` **implementations that they would rather use.**

(6) Precondition or postcondition?

- Whether to use a precondition, and if so, whether the method code should attempt to make sure the precondition has been met before proceeding? 是否应该使用前置条件？在方法正式执行之前，是否要检查前置条件已被满足？
- **For programmer:**
  - The most common use of preconditions is to demand a property precisely because it would be hard or expensive for the method to check it.

If to check a condition would make a method unacceptably slow, a precondition is often necessary.

不写Precondition，就要在代码内部check；若代价太大，在规约里加入precondition，把责任交给client

Precondition or postcondition?

- **For user:** A non-trivial precondition inconveniences clients, because they have to ensure that they don't call the method in a bad state (that violates the precondition); if they do, there is no predictable way to recover from the error.
- **So users of methods don't like preconditions.** 客户端不喜欢太强的**precondition**，不满足**precondition**的输入会导致失败。
  - Thus, Java API classes tend to specify (as a postcondition) that they throw unchecked exceptions when arguments are inappropriate. 惯用做法是：不限定太强的precondition，而是在postcondition中抛出异常：输入不合法
  - This makes it easier to find the bug or incorrect assumption in the caller code that led to passing bad arguments.
  - In general, it's better to **fail fast** , as close as possible to the site of the bug, rather than let bad values propagate through a program far from their original cause. 尽可能在错误的根源处fail，避免其大规模扩散

Precondition or postcondition?

- The key factors are the cost of the check (in writing and executing code), and the scope of the method. 衡量标准：检查参数合法性的代价多大?
- If it's only called locally in a class, the precondition can be discharged by carefully checking all the sites that call the method.
- If the method is public, and used by other developers, it would be less wise to use a precondition. Instead, like the Java API classes, you should throw an exception.

- 归纳：是否使用前置条件取决于(1) check的代价；(2) 方法的使用范围
  - 如果只在类的内部使用该方法(private)，那么可以不使用前置条件，在使用该方法的各个位置进行check——责任交给内部client；
  - 如果在其他地方使用该方法(public)，那么必须要使用前置条件，若client端不满足则方法抛出异常。

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Summary

Summary

- A specification acts as a crucial firewall between the implementor of a procedure and its client.
- It makes separate development possible: the client is free to write code that uses the procedure without seeing its source code, and the implementor is free to write the code that implements the procedure without knowing how it will be used.

Summary

- **Safe from bugs**
  - A good specification clearly documents the mutual assumptions that a client and implementor are relying on. Bugs often come from disagreements at the interfaces, and the presence of a specification reduces that.
  - Using machine-checked language features in your spec, like static typing and exceptions rather than just a human-readable comment, can reduce bugs still more.
- **Easy to understand**
  - A short, simple spec is easier to understand than the implementation itself, and saves other people from having to read the code.
- **Ready for change**
  - Specs establish contracts between different parts of your code, allowing those parts to change independently as long as they continue to satisfy the requirements of the contract.

Summary

- Declarative specifications are the most useful in practice.
- Preconditions (which weaken the specification) make life harder for the client, but applied judiciously they are a vital tool in the software designer's repertoire, allowing the implementor to make necessary assumptions.

Summary

- **Safe from bugs.**
  - Without specifications, even the tiniest change to any part of our program could be the tipped domino that knocks the whole thing over.
  - Well-structured, coherent specifications minimize misunderstandings and maximize our ability to write correct code with the help of static checking, careful reasoning, testing, and code review.
- **Easy to understand**
  - A well-written declarative specification means the client doesn't have to read or understand the code.
- **Ready for change**
  - An appropriately weak specification gives freedom to the implementor, and an appropriately strong specification gives freedom to the client.
  - We can even change the specs themselves, without having to revisit every place they're used, as long as we're only strengthening them: weakening preconditions and strengthening postconditions.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

May 24, 2021