哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 3: Abstract Data Type (ADT) and Object-Oriented Programming (OOP)

3.2 Designing Specification

Ming Liu

March 19, 2018

Outline

Objective of this lecture

- **Understand preconditions and postconditions in method specifications, and be able to write correct specifications**
- **What preconditions and postconditions are, and what they mean for the implementor and the client of a method.**
- **Understand underdetermined specs, and be able to identify and assess nondeterminism**
- **Understand declarative vs. operational specs, and be able to write declarative specs**
- **Understand strength of preconditions, postconditions, and specs; and be able to compare spec strength**
- **Be able to write coherent, useful specifications of appropriate strength**

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

Return Values

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

`void` means “no type”

Variable Scope

```
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);
    }
}
```

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"**

A complete method

```
public class Hailstone {
  /**
   * Compute a hailstone sequence.
   * @param n  Starting number for sequence.  Assumes n > 0.
   * @return hailstone sequence starting with n and ending with 1.
   */
  public static List<Integer> hailstoneSequence(int n) {
    List<Integer> list = new ArrayList<Integer>();
    while (n != 1) {
        list.add(n);
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    list.add(n);
    return list;
  }
}
```

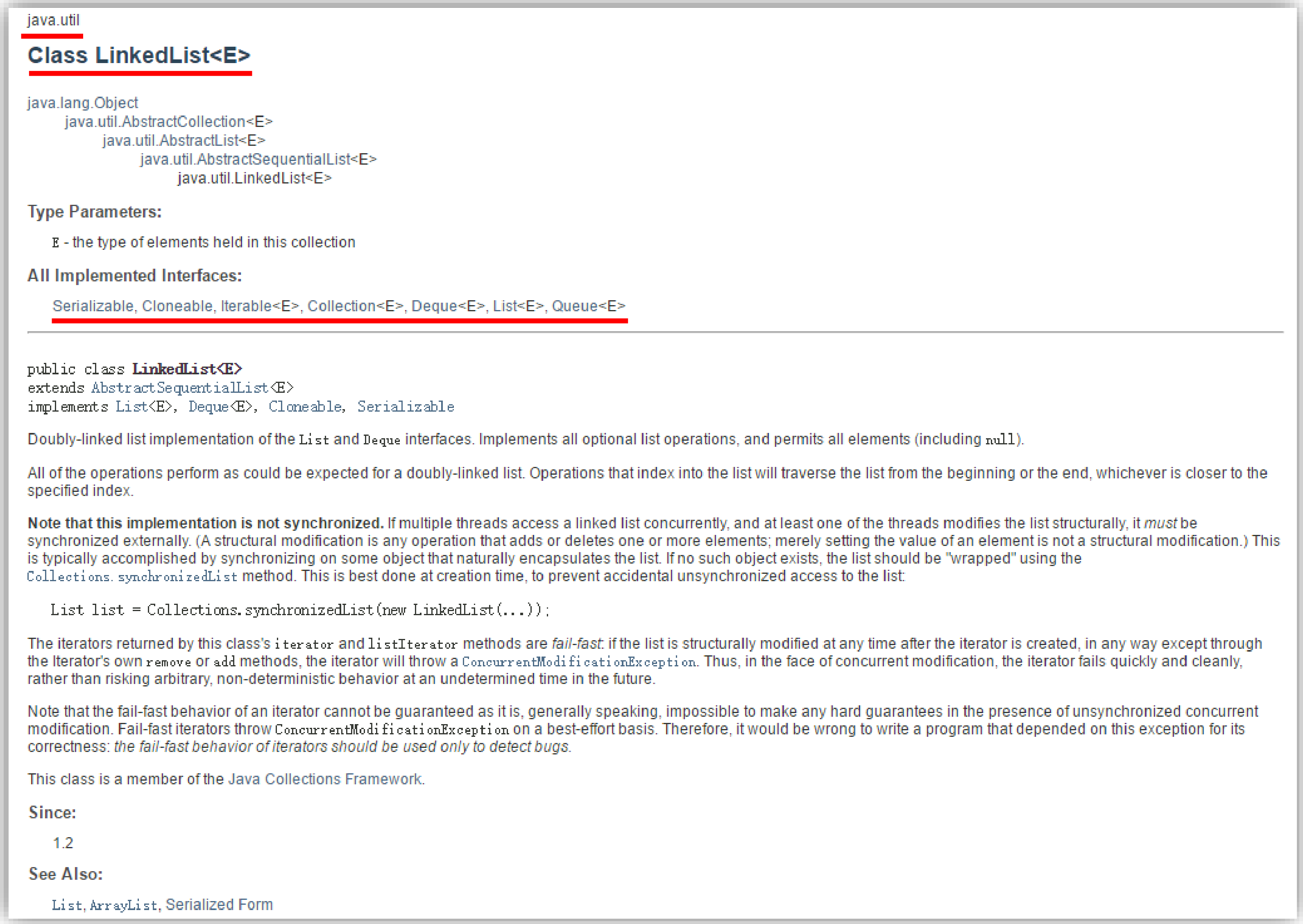
哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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.

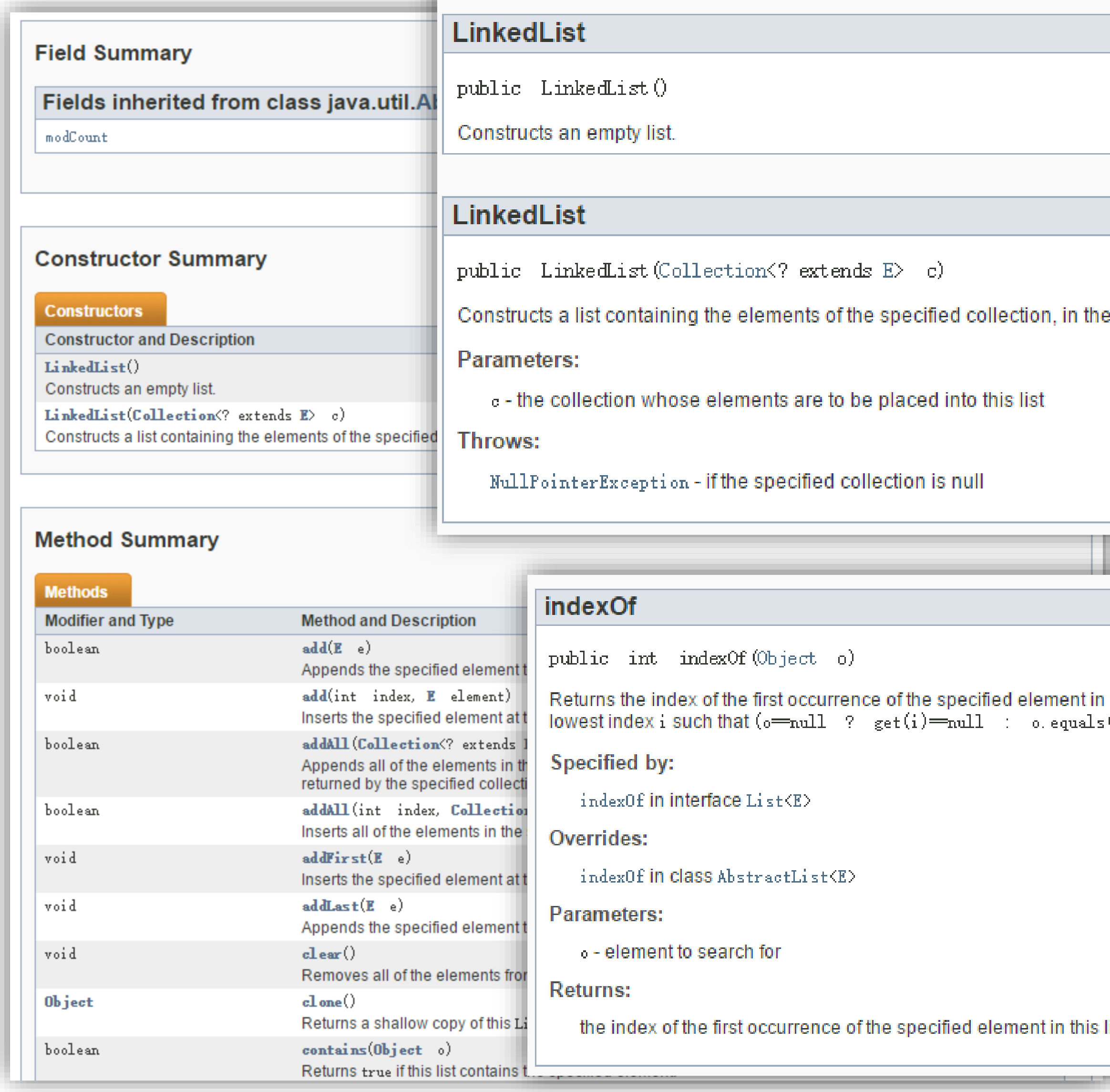
Since:

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

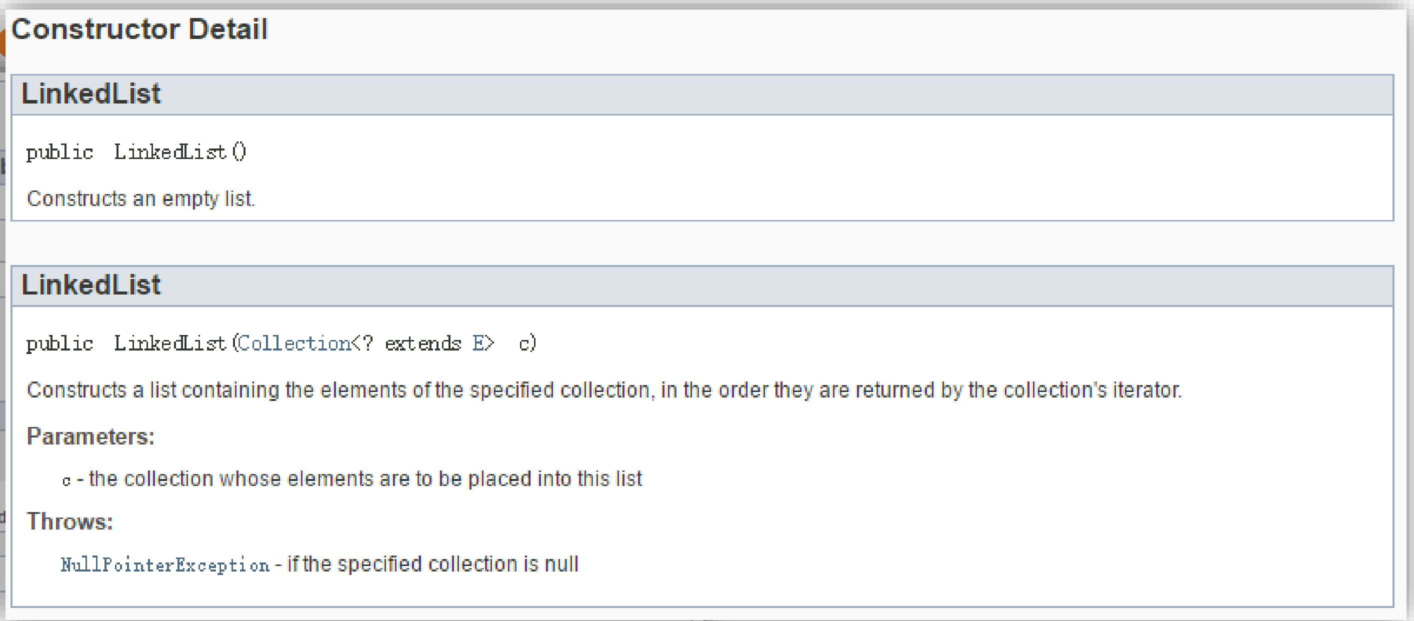
Constructors

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 I` 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 fro
`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)`

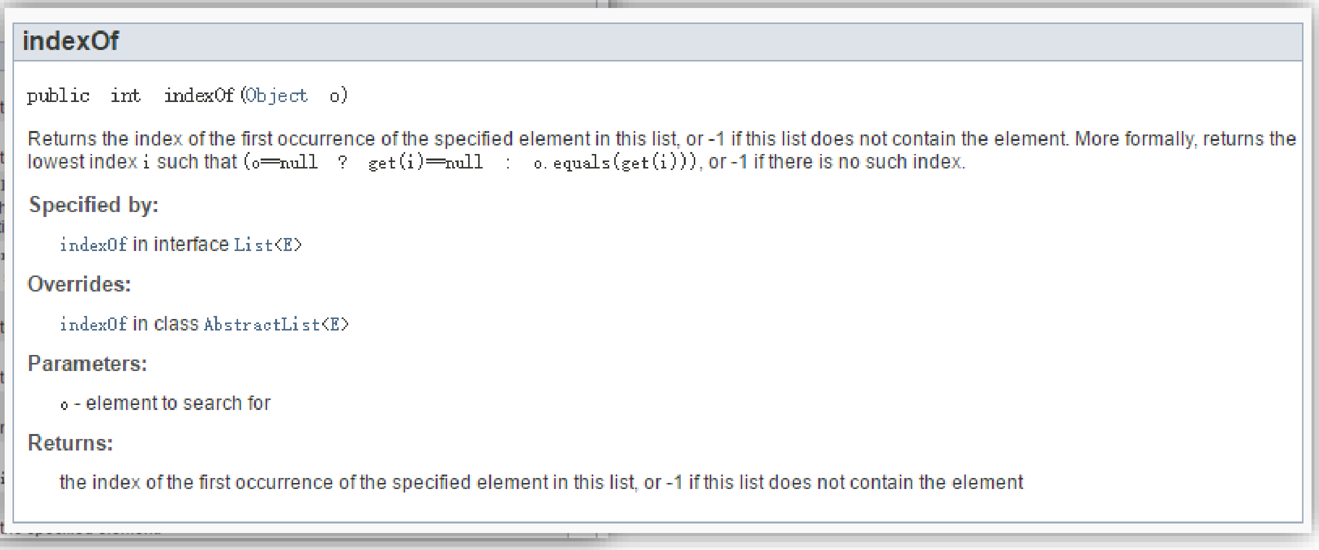
Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

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.**
  - 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.**
  - Java checks that too, statically.

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.

- **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.
  - 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.

Hacking vs. Engineering

- **Hacking is often marked by unbridled optimism:**
  - Bad: writing lots of code before testing any of it
  - Bad: keeping all the details in your head, assuming you'll remember them forever, instead of writing them down in your code
  - Bad: assuming that bugs will be nonexistent or else easy to find and fix
- **But software engineering is not hacking. Engineers are pessimists:**
  - Good: write a little bit at a time, testing as you go (test-first programming in Chapter 7).
  - Good: document the assumptions that your code depends on
  - Good: defend your code against stupidity – especially your own! Static checking helps with that.

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 of teamwork. It's impossible to delegate responsibility for implementing a method without a specification.**
- **The specification acts as a contract: the implementer is responsible for meeting the contract, and a client that uses the method can rely on the contract.**
  - States method's and caller's responsibilities
  - Defines what it means for implementation to be correct
- **Like real legal contracts, specifications place demands on both parties: when the specification has a precondition, the client has responsibilities too.**
  - If you pay me this amount on this schedule…
  - I will build a with the following detailed specification
  - Some contracts have remedies for nonperformance

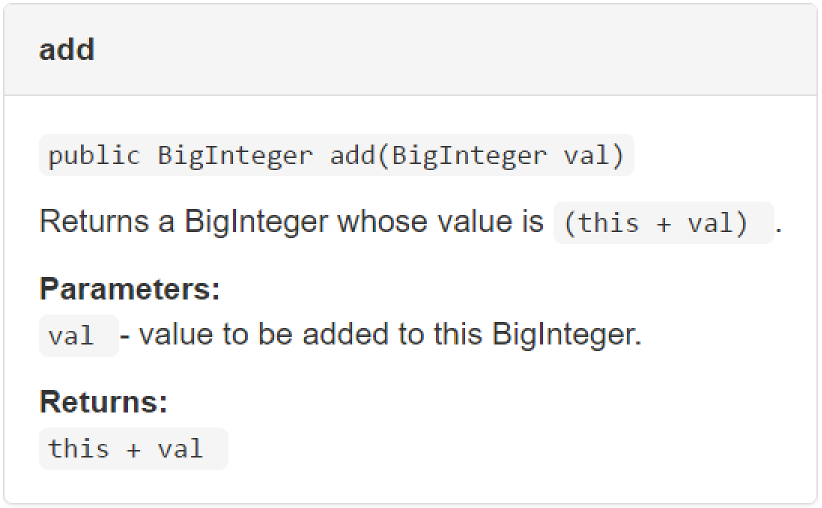
Why specifications?

- **Reality:**
  - Many of the nastiest bugs in programs arise because of misunderstandings about behavior at the interface between two pieces of code.
  - 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.
  - When the program fails, it's hard to determine where the error is.
- **Advantages:**
  - Precise specifications in the code let you apportion blame (to code fragments, not people!), and can spare you the agony of puzzling over where a fix should go.
  - Specifications are good for the client of a method because they spare the task of reading code.

An example of specification

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

Specification from the API documentation :



add

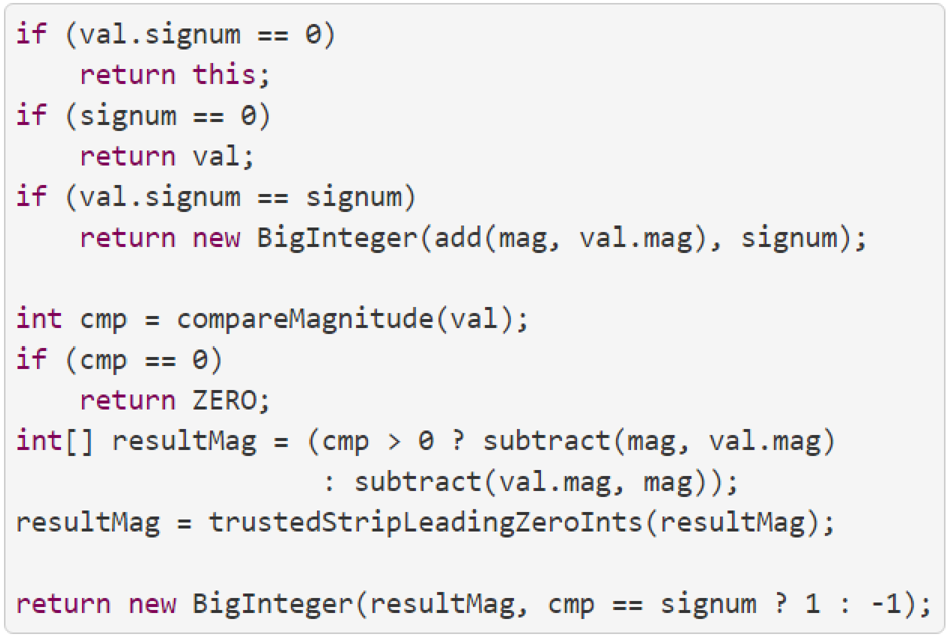
`public BigInteger add(BigInteger val)`

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 :



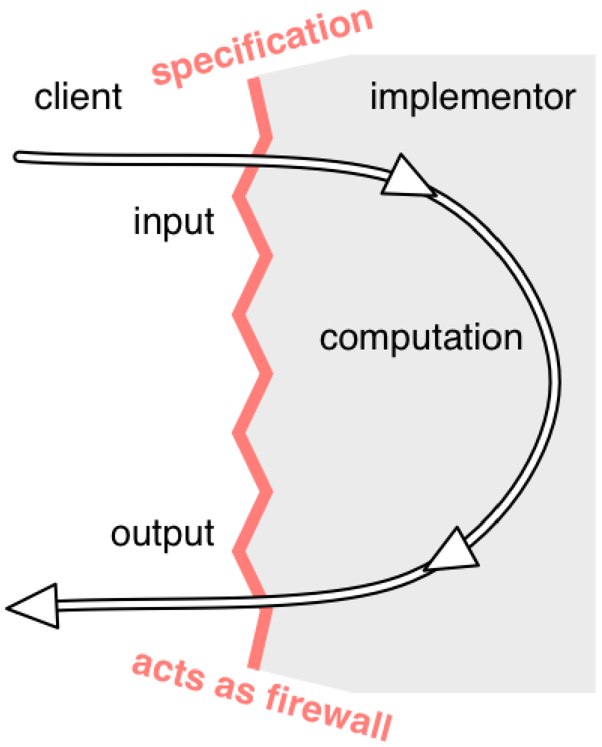
```java
if (val.signum == 0)
    return this;
if (signum == 0)
    return val;
if (val.signum == signum)
    return new BigInteger(add(mag, val.mag), signum);

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.
- Specifications can **make code faster**, too.
- **The contract acts as a *firewall* between client and implementor.**
  - 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.



(3) Behavioral equivalence

Behavioral equivalence

- To determine **behavioral equivalence** , the question is whether we could substitute one implementation for the other.

```
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;
}
```

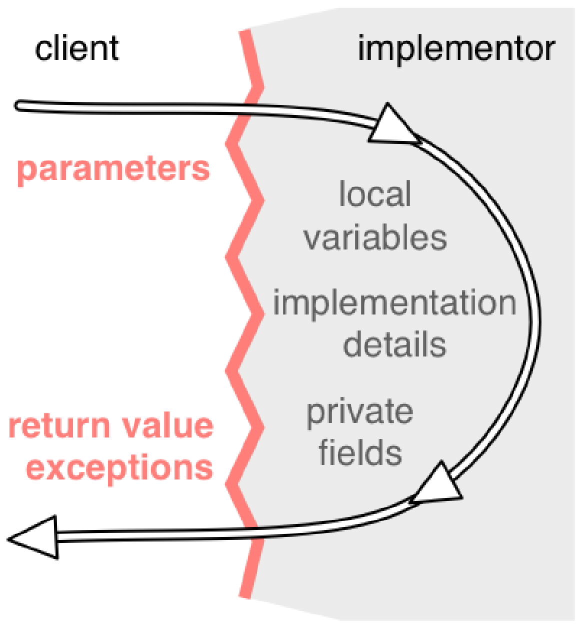
- The notion of equivalence is **in the eye of the client**.

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.

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

- Note: specification should never talk about local variables of the method or private fields of the method's class.



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

Specification Structure

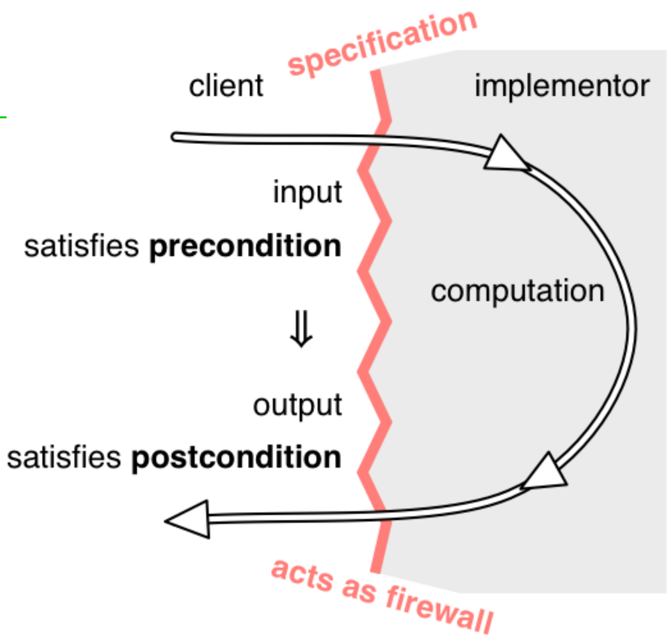
- **A specification of a method consists of several clauses:**
  - ***Precondition*** , indicated by the keyword `requires`
  - ***Postcondition*** , indicated by the keyword `effects`
  - **Exceptional behavior**: what it does if precondition violated

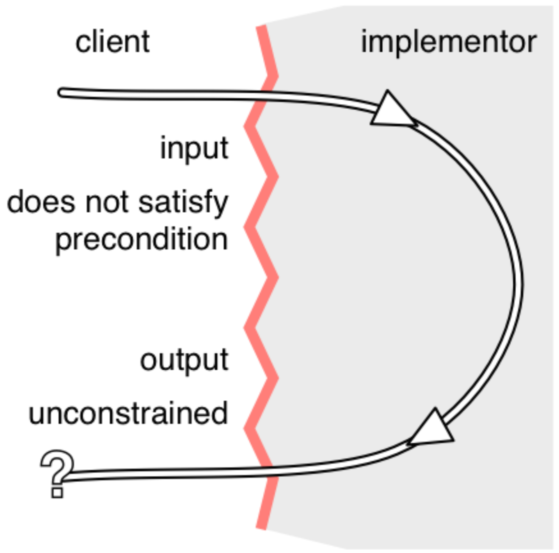
- The precondition is an obligation on the client (i.e., the caller of the method). It's a condition over the state in which the method is invoked.
- The postcondition is an obligation on the implementer of the method. If the precondition holds for the invoking state, the method is obliged to obey the postcondition, by returning appropriate values, throwing specified exceptions, modifying or not modifying objects, and so on.

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.

- 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.





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.
- 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.
- 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)
```

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.**
- **Virtually all programmers would assume the same thing. Surprise mutations lead to terrible bugs.**
- **Convention:**
  - Mutation is disallowed unless stated otherwise .
  - No mutation of the inputs
- **Mutable objects can make simple specification/contracts very complex**
- **Mutable objects reduce changeability**

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.
- **An example:** a method to looks up a username in database and returns the user's 9-digit identifier

```java
/**
 * @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:

```java
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:**

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

```
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

- **Sharing a mutable object complicates a contract.**
- **Who's to blame here?**
  - 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!

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?

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.

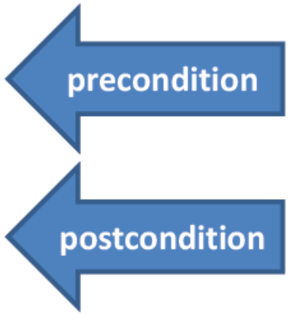
哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(5) Testing and verifying specifications

Formal contract specification

- **Java Modelling Language (JML)**

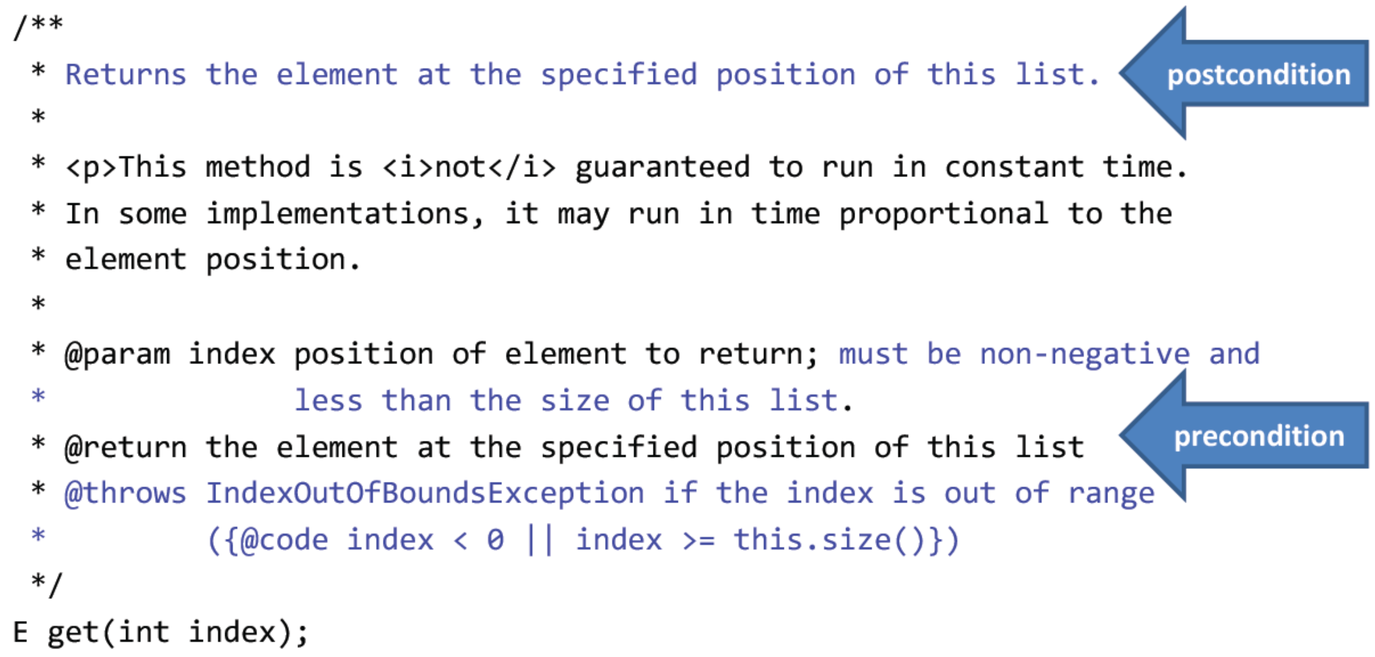
```
/*@ requires len >= 0 && array != null && array.length == len;
  @
  @ ensures \result ==
  @              (\sum int j;  0 <= j && j < len;  array[j]);
  @*/
int total(int array[], int len);
```



- **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);
```

Semantic correctness adherence to contracts

- **Compiler ensures types are correct (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*)**
  - – Overriding `equals` without overriding `hashCode`

Chapter 9 Refactoring

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 7 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.
- **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
```

3 Designing specifications

(1) Classifying specifications

Comparing specifications

- How **deterministic** 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?
- How **declarative** it is. Does the spec just characterize *what* the output should be, or does it explicitly say *how* to compute the output?
- How **strong** it is. Does the spec have a small set of legal implementations, or a large set?

- "What makes some specifications better than others?"

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
```

Deterministic vs. underdetermined specs

- **Under-deterministic:** specification allows multiple valid outputs for the same input.

```
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)
- To avoid the confusion, we'll refer to specifications that are not deterministic as **underdetermined** .
- 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.

Declarative vs. operational specs

- **Operational** specifications give a series of steps that the method performs; pseudocode descriptions are operational.
- **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.
- **Declarative specifications are preferable.**
  - They're usually shorter, easier to understand, and most importantly, they don't inadvertently expose implementation details that a client may rely on.
- **Why operational spec. exists?**
  - Programmers use the spec comment 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.

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
```

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?
- A specification S2 **is stronger than or equal to** a specification S1 if
  - S2's precondition is weaker than or equal to S1's
  - S2's postcondition is stronger than or equal to S1's, for the states that satisfy S1's precondition.

Then an implementation that satisfies S2 can be used to satisfy S1 as well, and it's safe to replace S1 with S2 in your program.

- **Rules:**
  - Weaken the precondition: placing fewer demands on a client will never upset them.
  - Strengthen the post-condition, which means making more promises.

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 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
```

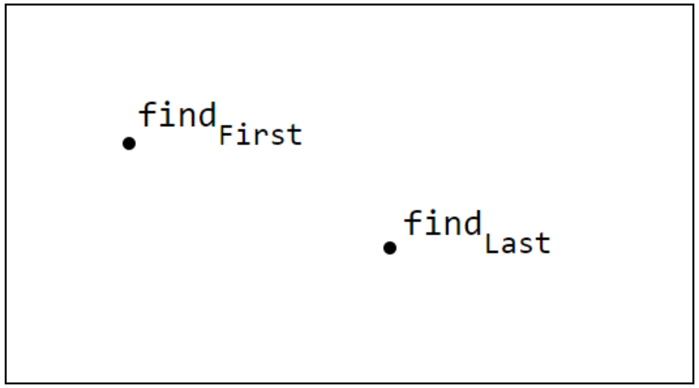
Stronger vs. weaker specs

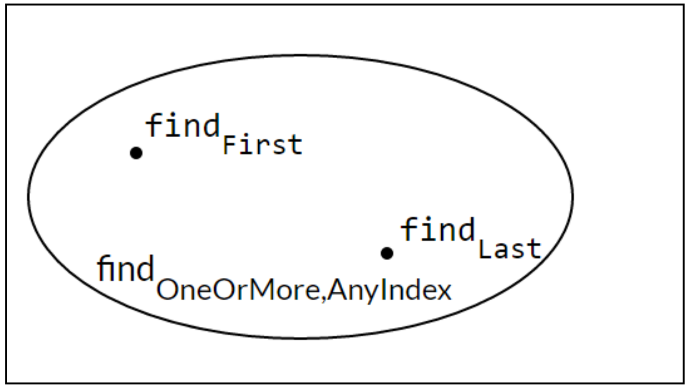
- If S3 is neither stronger nor weaker than S1, there specs. might overlap (such that there exist implementations that satisfy only S1, only S3, and both S1 and S3) or might be disjoint.
- In both cases, S1 and S3 are incomparable.

(2) Diagramming specifications

Diagramming specifications

- Each point in this space represents a method implementation.

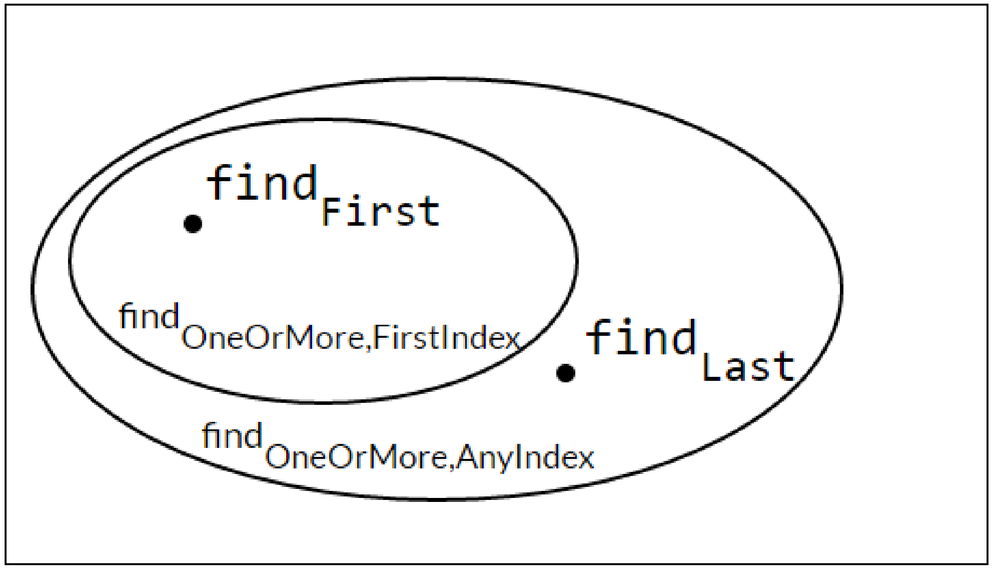




- A specification defines a *region* in the space of all possible implementations.
- A given implementation either behaves according to the spec, satisfying the precondition-implies-postcondition contract (it is inside the region), or it does not (outside the region).

Diagramming specifications

- When S2 is stronger than S1, it defines a *smaller* region in this diagram; a weaker specification defines a larger region.


$\text{find}_{\text{First}}$
$\text{find}_{\text{OneOrMore,FirstIndex}}$
$\text{find}_{\text{Last}}$
$\text{find}_{\text{OneOrMore,AnyIndex}}$

(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, 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 rules, but there are some useful guidelines.

The specification should be coherent

- The spec shouldn't have lots of different cases. Long argument lists, deeply nested if-statements, and boolean flags are all signs of trouble.

```
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)
```

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

The results of a call should be informative

```
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`.**

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. Here's a specification illustrating this flaw (and also written in an inappropriately operational style):

```
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
```

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

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.

The specification should use *abstract types*

- **Abstract notions like a `List` or `Set`**
- **Particular implementations like `ArrayList` or `HashSet`.**
- **Writing our specification with abstract types gives more freedom to both the client and the implementor.**
- **In Java, this often means using an interface type, like `Map` or `Reader`, instead of specific implementation types like `HashMap` or `FileReader`.**

```
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()
```

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?
- **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.

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.
  - Thus, Java API classes tend to specify (as a postcondition) that they throw unchecked exceptions when arguments are inappropriate.
  - 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.

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

March 19, 2018