Advanced Object-Oriented Programming

CPT204 – Lecture 5

Erick Purwanto

Xi'an Jiaotong-Liverpool University

西交利物浦大学

CPT204 Advanced Object-Oriented Programming

Lecture 5

Specification, Linked List 2

Welcome !

- Welcome to Lecture 5 !
- In this lecture we are going to
  - continue to learn about specification
    - preconditions and postconditions
    - testing against specs
  - continue improving our homemade linked list, while learning about
    - inner/nested class
    - caching
    - invariants
    - constant-time methods
    - generics

Part 1: Specification

- This week, we are going to continue to discuss specification, which we have seen every week in the **javadoc**

- A **specification** is like a contract for part of your program
  - saying what it can count on from the rest of the program,
  - and what it's expected to do in return

A Contract between Parties

- 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
  - In fact, we'll see that like real legal contracts, specifications place demands on ***both parties***: when the specification has a *precondition*, the client has *responsibilities* too!
- We'll look at the role played by specifications of methods
  - We'll discuss what preconditions and postconditions are, and what they mean for the implementer and the client of a method
  - We'll also talk about how to use **exceptions**, an important language feature found in Java which allows us to make a method's interface safer from bugs and easier to understand

Why Specification?

- 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 same team have different specifications in mind
  - When the program fails, it's hard to determine where the error is. 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*
  - for example, take a look at some of the standard Java specs
  - compare them to the source code that implements them

Example of Reading Spec vs Actual Code (1)

- Here's an example of one method from 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);
```

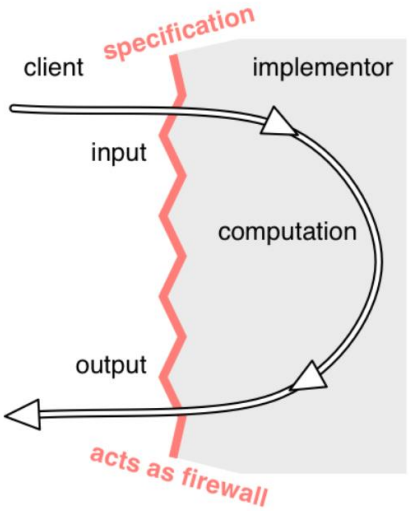
Example of Reading Spec vs Actual Code (2)

- The spec for `BigInteger.add` is straightforward for clients to understand, and if we have questions about corner cases, the `BigInteger` class provides additional human-readable documentation

- If all we had was the code, we'd have to read through the `BigInteger` constructor, `compareMagnitude`, `subtract`, and `trustedStripLeadingZeroInts` just as a starting point

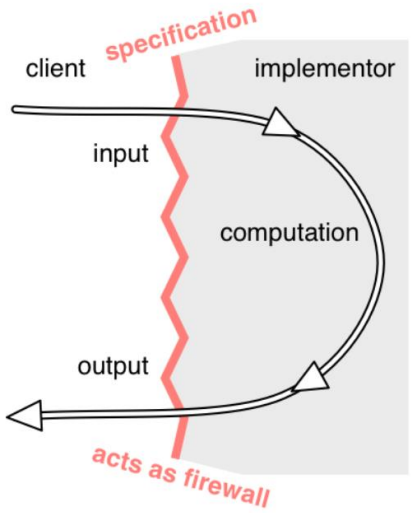
Spec and Implementer

- Specifications are good for the implementer of a method because they give the implementer *freedom to change* the implementation *without* telling clients
- Specifications can make code *faster*, too
  - We'll see that using a weaker specification can rule out certain states in which a method might be called
  - This restriction on the inputs might allow the implementer to skip an expensive check that is no longer necessary and use a more efficient implementation



Spec and Client

- The contract acts as **a firewall** between the client and the implementer
  - It shields the client from the details of the *workings* of the unit — you don't need to read the source code of the procedure if you have its specification
  - And it shields the implementer from the details of the *usage* of the unit
  - The implementer doesn't have to ask every client how they plan to use 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 — each obeying its obligation



Behavioral Equivalence (1)

- Consider these two methods: are they the same or different?

```
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;
}
```

Behavioral Equivalence (2)

- Of course the code is different, so in that sense they are different; and we've given them different names, just for the purpose of discussion
  - To determine behavioral equivalence, our question is *whether we could substitute one implementation for the other*

- Not only do these methods have different code, they actually have different behavior:
  - 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

Behavioral Equivalence (3)

- But when `val` occurs at exactly one index of the array,
  the two methods ***behave the same***: they both return that index
  - It may be that clients never rely on the behavior in the other cases: 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, and we could switch from one implementation to the other without issue!

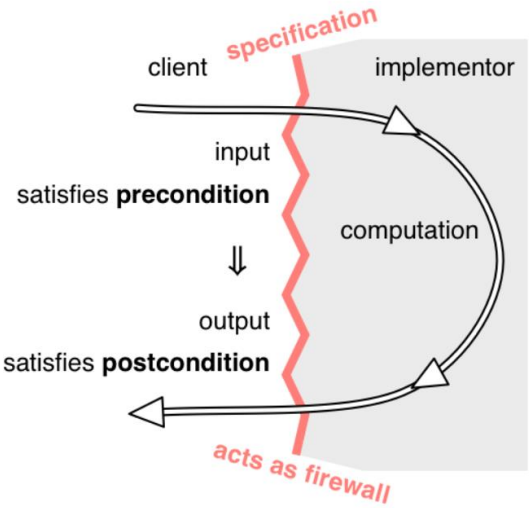
Behavioral Equivalence (4)

- The notion of equivalence is in the eye of the beholder — that is, *the client*
  - 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
  - In this case, our specification might be:

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

Specification Structure (1)

- A specification of a method consists of two clauses:
  - ***a precondition***, indicated by the keyword **requires**
  - ***a postcondition***, indicated by the keyword **effects**

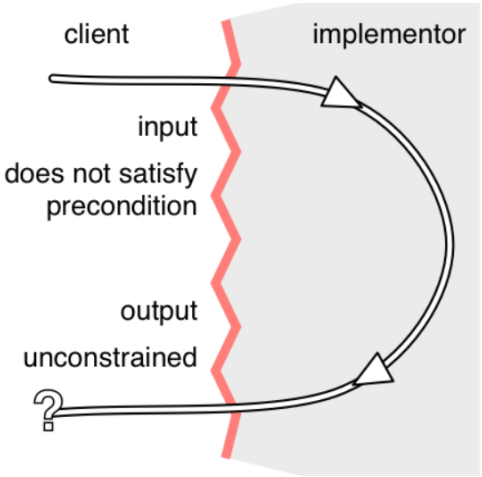


- The precondition is an obligation on the client (i.e., the caller of the method) — it's a condition over the state in which the method is invoked
- The postcondition is an obligation on the implementer of the method
- If the precondition holds for the invoking state, the method is obliged to obey the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

Specification Structure (2)

- 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



In-Class Quiz 1

- Given the following specification :

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

select the **legal** behavior that you can then implement `find` with:

- ❑ if `arr` is empty, return `0`
- ❑ if `val` occurs twice in `arr`, set all values in `arr` to zero, then throw an exception
- ❑ if `val` does not occur in `arr`, pick random `index`, set value at `index` to `val`, return the `index`
- ❑ if `arr[0]` is `val`, continue search, if found another return the index; otherwise return `0`

Specifications in Java (1)

- Some languages (notably Eiffel) incorporate preconditions and postconditions as a fundamental part of the language
  - as expressions that the runtime system (or even the compiler) can automatically check to enforce the contracts between clients and implementers
- Java does *not* go quite so far, but its 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 — the parts that we can't write as types — must be described in ***a comment*** preceding the method, and generally depends on *human* to check it and guarantee it

Specifications in Java (2)

- Java has a convention for documentation comments, in which parameters are described by **`@param`** clauses and results are described by **`@return`** and **`@throws`** clauses
  - Put the preconditions into **`@param`** where possible
  - Put postconditions into **`@return`** and **`@throws`**

Specifications in Java (3)

- A specification like this:

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

might be rendered in Java like this:

```
/**
 * 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)
```

In-Class Quiz 2

- Given the following specification :

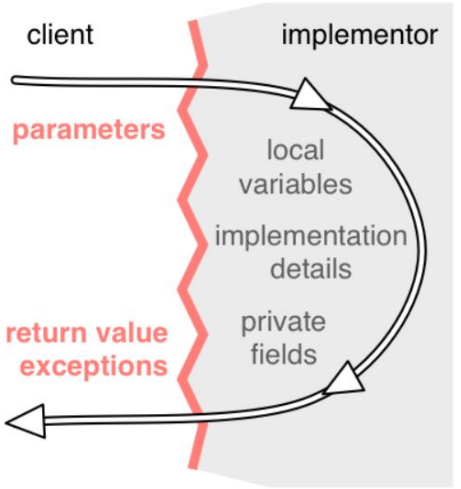
```
static boolean isPalindrome(String word)
  requires: word contains only alphanumeric characters
  effects:  returns true if and only if word is a palindrome
```

which line of the Javadoc comment is ***problematic***:

```
/**
 * Check if a word is a palindrome.                                    (1)
 * A palindrome is a sequence of characters                            (2)
 * that reads the same forwards and backwards.                         (3)
 * @param String word to check, must contain only alphanumeric characters  (4)
 * @return true if and only if word is a palindrome                    (5)
 */
```

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
  - 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



Testing and Specifications (1)

- In lecture about Testing, we talk about black box tests that are chosen with only the specification in mind, and glass/white box tests that are chosen with knowledge of the actual implementation
  - but it's important to note that even glass box tests **must follow** the specification
  - your implementation may provide stronger guarantees than the specification calls for, or it may have specific behavior where the specification is undefined
  - but your test cases should not count on that behavior
  - test cases must obey the contract, just like every other client

- For example, suppose you are testing this specification of `find`, which is slightly different from the one we've used so far:

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

Testing and Specifications (2)

- This spec has a strong precondition in the sense that val is required to be found; and it has a fairly weak postcondition in the sense that if val appears more than once in the array, this specification ***says nothing about which particular index of val is returned***
- Even if you implemented find so that it always returns the lowest index, your test case **can't assume** that specific behavior:

```
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
```

Testing and Specifications (3)

- Similarly, even if you implemented find so that it (sensibly) throws an exception when `val` isn't found, instead of returning some arbitrary misleading index,
  - your test case can't assume that behavior, because it can't call `find()` in a way that violates the precondition
- So what does glass box testing mean, if it can't go beyond the spec?
  - It means you are trying to find new test cases that *exercise different parts of the implementation*, but still checking those test cases in an *implementation-independent* way

Testing Units (1)

- Recall the web search example from last week lecture with these methods:

```
/** @return the contents of the web page downloaded from url */
public static String getWebPage(URL url) { ... }

/** @return the words in string s, in the order they appear,
 *          where a word is a contiguous sequence of
 *          non-whitespace and non-punctuation characters */
public static List<String> extractWords(String s) { ... }

/** @return an index mapping a word to the set of URLs
 *          containing that word, for all webpages in the input set */
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {
    ...
    calls getWebPage and extractWords
    ...
}
```

Testing Units (2)

- We talked then about ***unit testing***, the idea that we should write tests of each module of our program in isolation
  - A good unit test is focused on just a single specification
  - Our tests will nearly always rely on the specs of Java library methods, but a unit test for one method we've written *shouldn't fail* if a different method fails to satisfy its spec
  - As we saw in the example, a test for `extractWords` shouldn't fail if `getWebPage` doesn't satisfy its postcondition

Testing Units (3)

- Good **integration tests**, tests that use a combination of modules, will make sure that our different methods have compatible specifications: callers and implementers of different methods are passing and returning values as the other expects
  - Integration tests cannot replace systematically-designed unit tests
  - From the example, if we only ever test `extractWords` by calling `makeIndex`, we will only test it on a potentially small part of its input space: inputs that are possible outputs of `getWebPage`
  - In doing so, we've left a place for bugs to hide, ready to jump out when we use `extractWords` for a different purpose elsewhere in our program, or when `getWebPage` starts returning web pages written in a new format, etc

Specifications for Mutating Methods (1)

- We previously discussed mutable vs. immutable objects, but our specifications of find didn't give us the opportunity to illustrate how to describe side-effects — changes to mutable data — in the postcondition
- Here's a specification that describes a method that mutates an object:

```
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
```

- First, look at the postcondition
  - It gives two constraints: the first telling us how `list1` is modified, and
  - the second telling us how the return value is determined

Specifications for Mutating Methods (2)

- The specification from previous slides:

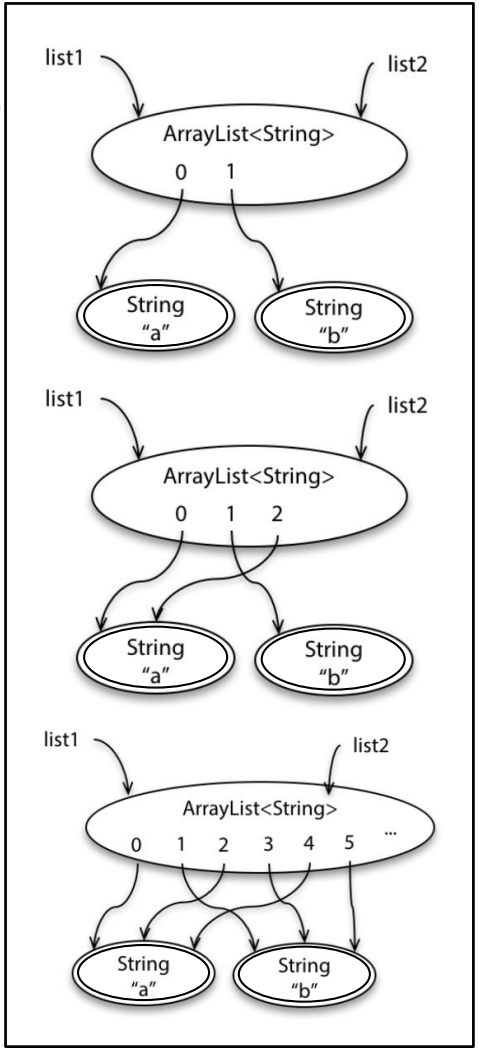
> `static boolean` **`addAll`**`(List<T> list1, List<T> list2)`
> **requires:** `list1 != list2`
> **effects:** modifies `list1` by adding the elements of `list2` to the end of it, and returns `true` if `list1` changed as a result of call

- Second, look at the precondition — it tells us that the behavior of the method if you attempt to add the elements of a list to itself is undefined
  - you can easily imagine why the implementer of the method would want to impose this constraint:
    - it's not likely to rule out any useful applications of the method,
    - and it makes it easier to implement

Specifications for Mutating Methods (3)

- The specification allows a simple algorithm in which you take an element from `list2` and add it to `list1`, then go on to the next element of `list2` until you get to the end of `list2`

- If `list1` and `list2` are the same list, then both lists will keep growing, and you will never get to the end
  - the ***sequence of snapshot diagrams at right*** illustrate this behavior
  - the simple algorithm above will not terminate — or practically speaking it will throw a memory error when the list object has grown so large that it consumes all available memory
  - either outcome, infinite loop or crash, is *permitted* by the specification because of its precondition



Specifications for Mutating Methods (3)

- Remember also our implicit precondition that `list1` and `list2` must be valid objects, rather than `null`
- We'll usually *omit* saying this because it's virtually always required of object references

- Here is another example of a *mutating* method:

```
static void sort(List<String> list)
```
requires: nothing

effects: puts list in sorted order, i.e. `list[i] <= list[j]` for all `0 <= i < j < list.size()`

- And an example of a method that *does not mutate* its argument:

```
static List<String> toLowerCase(List<String> list)
```
requires: nothing

effects: returns a new list `t` where `t[i] = list[i].toLowerCase()`

Specifications for Mutating Methods (4)

- Just as we've said that `null` is implicitly disallowed unless stated otherwise, we will also use the convention that ***mutation is disallowed*** unless stated otherwise
  - The spec of `toLowerCase` could explicitly state as an effect that “list is not modified”, but in the absence of a postcondition describing mutation, we demand no mutation of the inputs

Part 2: From MyList to Singly Linked List (SLList)

- In the next part of lecture, we will improve our homemade list
- We want users to use our data structure more easily, hiding details from them

```
MyList2 list = new MyList2( value: 10, next: null);
list = new MyList2( value: 2, list);
list = new MyList2( value: 5, list);
```

- We want users to create and add data without needing to specify reference/pointer

```
SLList1 list = new SLList1( item: 10);
list.addFirst( item: 2);
list.addFirst( item: 5);
```

Inner / Nested Class

- For our new first implementation of list `SLList1`, we use an inner class called Node, to store the **item** and pointer to the **next** node in the list (or `null`, if it's the last node)

```
private static class Node {
    public int item;
    public Node next;

    public Node(int i, Node n) {
        item = i;
        next = n;
    }
}
```

class Node is defined **inside** class SLList

private, since Node is only used by SLList, and not other classes

static, since Node never uses instance variables/methods of the outer class

for private inner classes, these access modifiers are irrelevant

Instance Variable and Constructor

- We store the pointer to the first node and always start with a list of one item

```
// The first node at the front of the list
private Node first;

/** Creates an SLList with one item. */
public SLList1(int item) {
    first = new Node(item, n: null);
}
```

AddFirst and GetFirst

- addFirst is used to store a new item at the beginning of the list
  - while getFirst returns the item stored in the first node of the list

```java
/**
 * Adds a new node with an input item to the front of the list.
 * @param item is an integer.
 */
public void addFirst(int item) {
    first = new Node(item, first);
}

/**
 * @return the first item in the front of the SLList.
 */
public int getFirst() {
    return first.item;
}
```

this will be enough to give what we want at the beginning

```java
SLList1 list = new SLList1( item: 10);
list.addFirst( item: 2);
list.addFirst( item: 5);
```

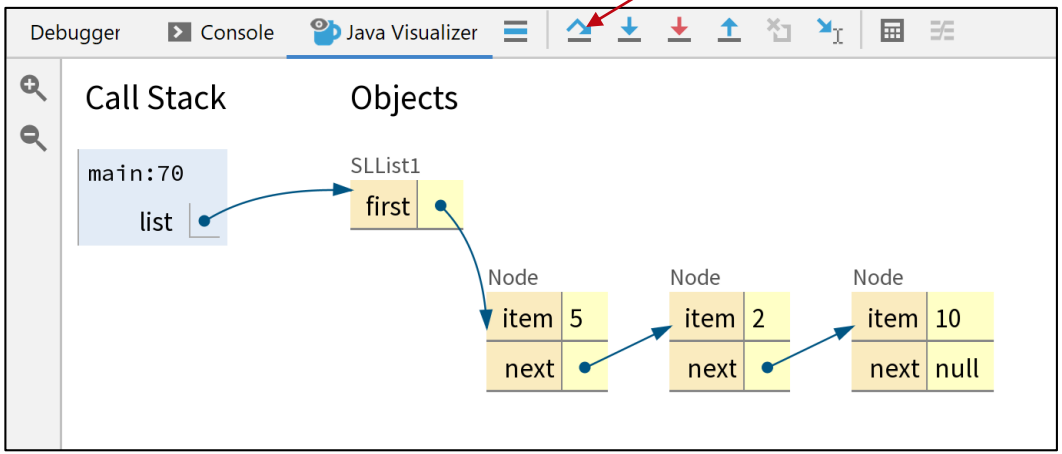
but let's add more functionalities to our list

Java Visualizer

- Recall in week 4, you can use Java Visualizer to help you understand what's going on

```
66    public static void main(String[] args) {  args: {}
67        SLList1 list = new SLList1( item: 10);  list: SLList1@790
68        list.addFirst( item: 2);
69        list.addFirst( item: 5);
```

use Step Over to skip going into a method



AddLast

- To add item at the end of a list, we have to go to the last node first

```
/**
 * Adds item to the end of the list.
 * @param item is an integer.
 */
public void addLast(int item) {
    Node p = first;
    // Advance p to the end of the list
    while (p.next != null) {
        p = p.next;
    }
    p.next = new Node(item,  n: null);
}
```

Size

- To return the size recursively, we have to use a recursive helper method

```java
/**
 * @return the size of the SLList recursively.
 */
public int size() { return size(first); }

// Recursive size helper method
private int size(Node p) {
    // base case
    if (p.next == null) {
        return 1;
    }
    // recursive step
    return 1 + size(p.next);
}
```

stop!

this implementation of list takes *linear time* to return the size

can you think of a way to return the size in **constant time**?

SLList2 with Caching Size Instance Variable

- We store the size information as an instance variable, set and update it accordingly,

```
// The first node at the front of the list
private Node first;
private int size;

/** Creates an SLList with one item. */
public SLList2(int item) {
    first = new Node(item, n: null);
    size = 1;
}
```

```
public void addFirst(int item) {
    first = new Node(item, first);
    size += 1;
}
```

so can return size in constant time:

```
/**
 * @return the size of the SLList.
 */
public int size() {
    return size;
}
```

this technique is called **caching**
we can also use caching to store max, min, mode

Empty SLList

- Suppose we want to start with no item. How do we create an empty list?

```
/** Creates an empty SLList with no item. */
public SLList2() {
    first = null;
    size = 0;
}
```

- It works well with:

```
SLList2 list2 = new SLList2();
list2.addFirst( item: 200);
list2.addFirst( item: 100);
list2.addLast( item: 300);
System.out.println(list2.size());
```

stop!

it turns out there is ***a bug***.
can you think how to add to the list starting with empty list that can cause an error?

Bug in Starting with an Empty SLList

- When you do addLast immediately after creating an empty SLList, there is a runtime error

```
SLList2 list3 = new SLList2();
list3.addLast( item: 50);
list3.addFirst( item: 72);
```

```
Exception in thread "main" java.lang.NullPointerException
    at SLList2.addLast(SLList2.java:52)
    at SLList2.main(SLList2.java:81)
```

- In addLast:

```
public void addLast(int item) {
    Node p = first;
    // Advance p to the end of the list
    while (p.next != null) {
        p = p.next;
```

we cannot have **null.**next !

stop!

how do you fix this bug in addLast?

AddLast v2

- We modify `addLast` so that it simply adds a node if it was empty

```java
/**
 * Adds item to the end of the list.
 * @param item is an integer.
 */
public void addLast(int item) {
    if (first == null) {
        first = new Node(item, n: null);
        return;
    }
    Node p = first;
    // Advance p to the end of the list
    while (p.next != null) {
        p = p.next;
    }
    p.next = new Node(item, n: null);
    size += 1;
}
```

add the following code

there is another solution

SLList3 with Sentinel Node: Empty Constructor

- Motivation: we want to avoid special cases, where an empty list is null
- Idea: let the empty list be a list with one sentinel node, a node without real data

```
// The first node at the front of the list is a sentinel node
// The first item, if it exists, is in the sentinel.next node
private Node sentinel;
private int size;
```

```
/** Creates an empty SLList with no item. */
public SLList3() {
    sentinel = new Node( i: 0, n: null);
    size = 0;
}
```

instead of null,
now an empty SLList contains
a sentinel node

the item here can be anything
it does not matter

so now, the list is never null

SLList with Sentinel: Constructor and AddFirst

- The constructor for a one-item list:

```
/** Creates an SLList with one item. */
public SLList3(int item) {
    sentinel = new Node( i: 0,  n: null);
    sentinel.next = new Node(item,  n: null);
    size = 1;
}
```

create a sentinel node,
and the real node with input data

- Adding in the front of the list:

```
public void addFirst(int item) {
    sentinel.next = new Node(item, sentinel.next);
    size += 1;
}
```

create a new node in between the sentinel and the first real node

SLList with Sentinel: GetFirst and AddLast

- Getting the first item of the list:

```
public int getFirst() {
    return sentinel.next.item;
}
```

the first item is in the node after the sentinel

- Adding in the end of the list:

```
public void addLast(int item) {
    Node p = sentinel;
    // Advance p to the end of the list
    while (p.next != null) {
        p = p.next;
    }
    p.next = new Node(item, n: null);
    size += 1;
}
```

no special cases,
just find the end of list to add node,
which could be after the sentinel

Invariants

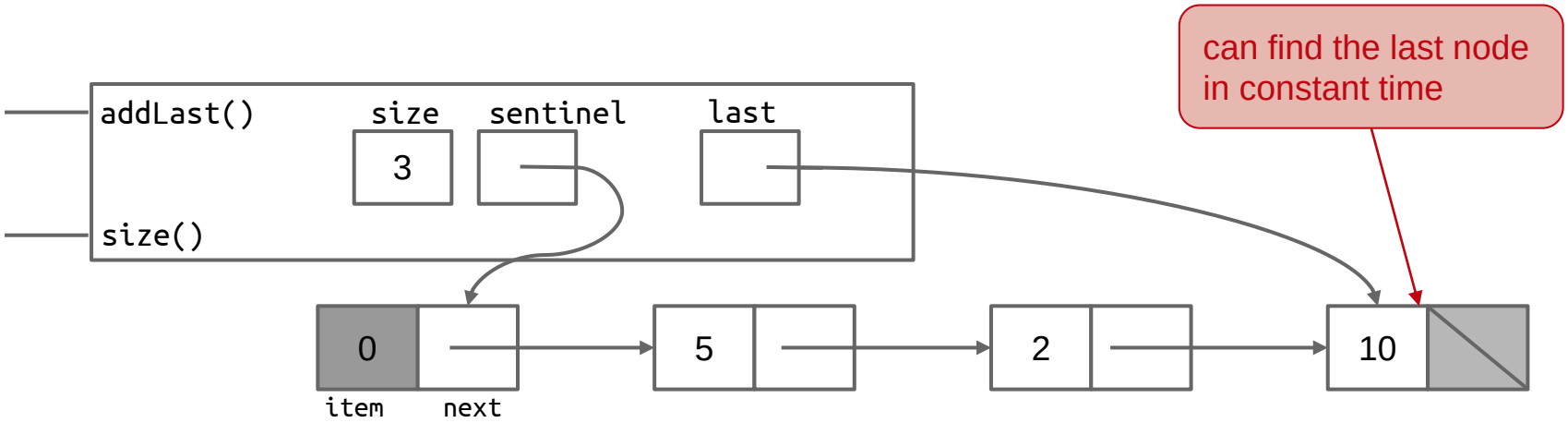
- **An invariant** is a condition that is guaranteed to be ***true*** during code execution

- For example, `SLList` with Sentinel Node has the following invariants:
  - sentinel instance variable *always* points to a sentinel node
  - the first node, if it exists, is *always* at sentinel.next
  - size instance variable is *always* the total number of items added

- Invariants make it easier to reason about code:
  - can assume they are true to simplify code,
    e.g., addLast does not need to worry about the null case
  - must ensure that the methods ***preserve invariants***

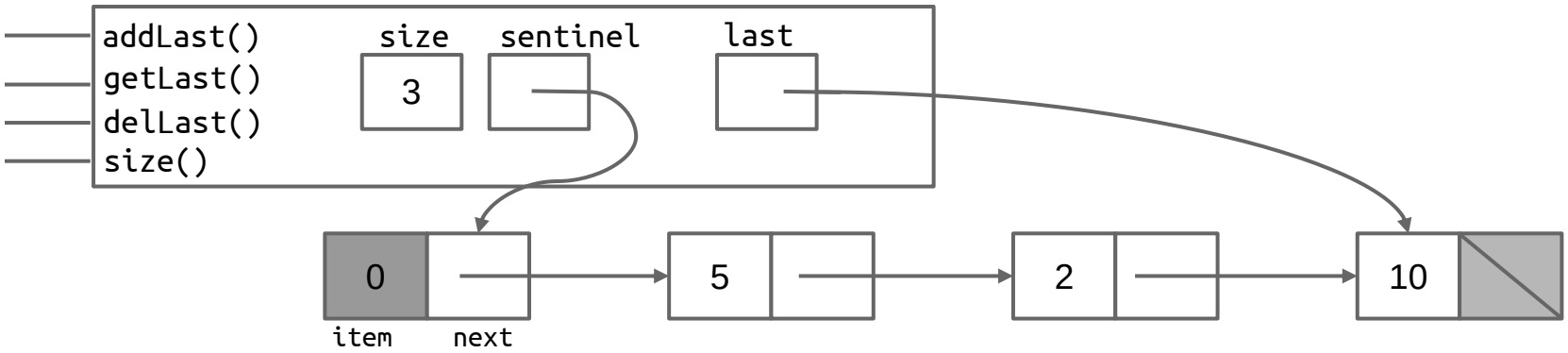
Next Improvement: AddLast

- AddLast is slow, taking linear time, proportional to size of list
  - What should we do to make it fast?
- We may think, using the caching technique like last time, store a pointer to the last node



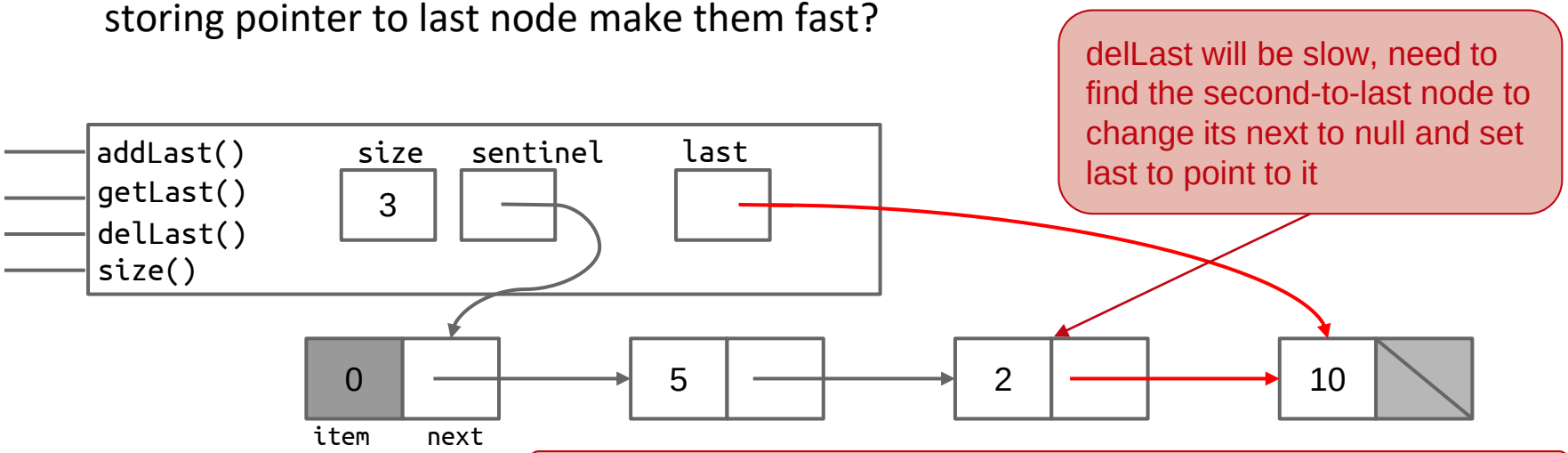
Next Improvement: AddLast, GetLast, DelLast

- AddLast is slow, taking linear time, proportional to size of list
  - What should we do to make it fast?
- We may think, using the caching technique like last time, store a pointer to the last node
  - Suppose now we want to **addLast**, **getLast**, and **delLast** from the back, would storing pointer to last node make them fast?



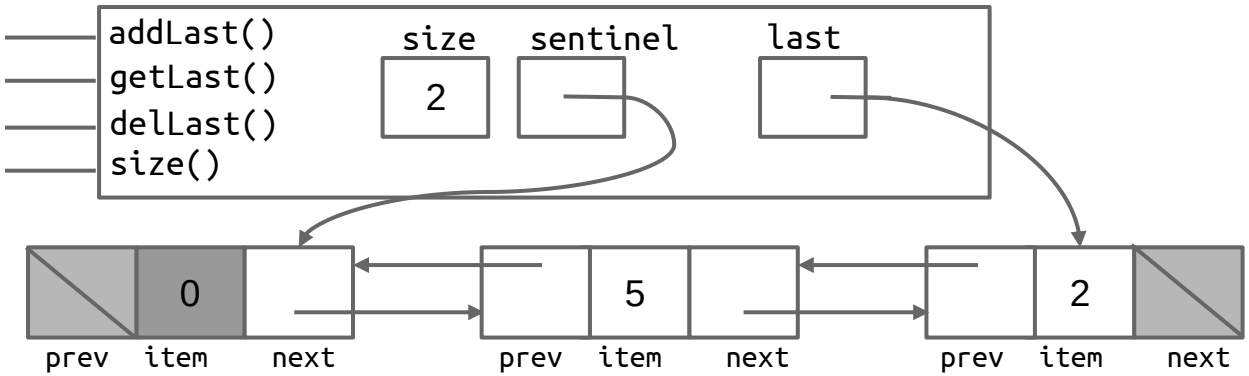
Next Improvement: AddLast, GetLast, DelLast

- AddLast is slow, taking linear time, proportional to size of list
  - What should we do to make it fast?
- We may think, using the caching technique like last time, store a pointer to the last node
  - Suppose now we want to **addLast**, **getLast**, and **delLast** from the back, would storing pointer to last node make them fast?



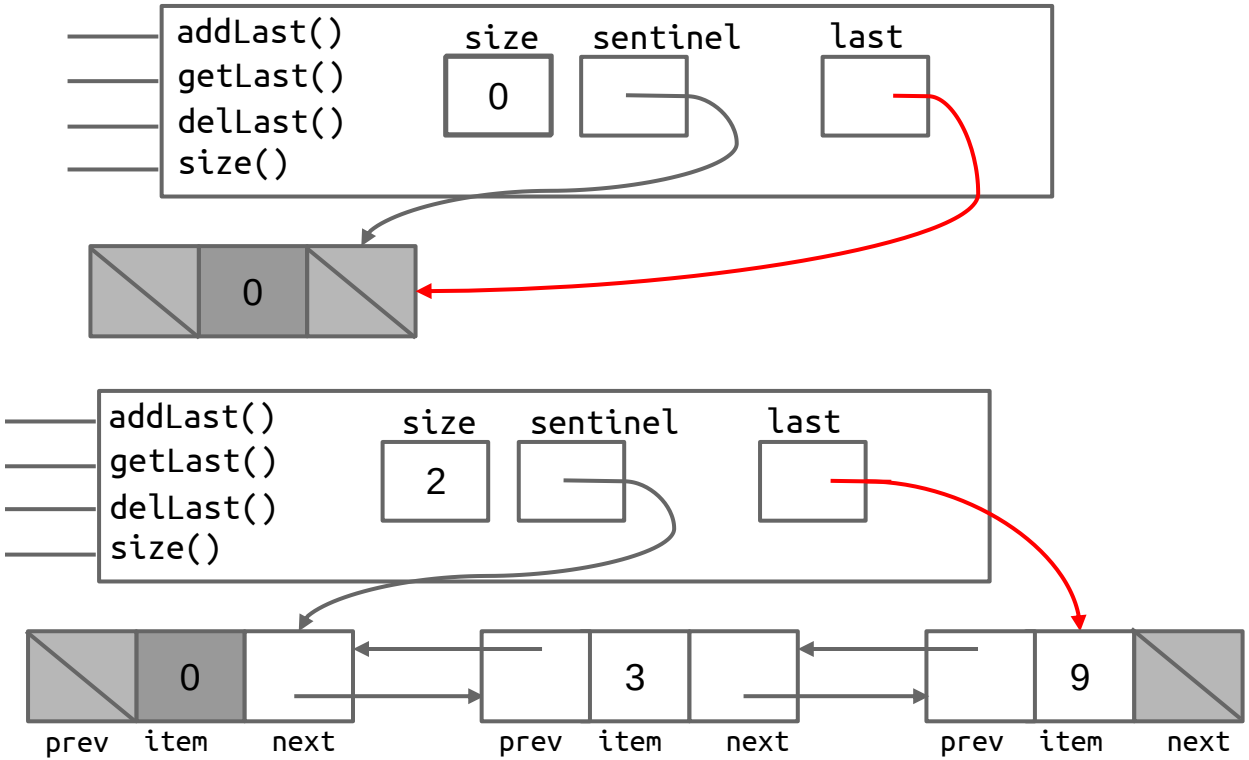
Doubly-Linked List

- In addition to a pointer to the last node, to make removeLast fast, need to add backward link from every node to its previous node, let's call it prev
- This results in a Doubly-Linked List (DLList) :



Doubly-Linked List with Single Sentinel

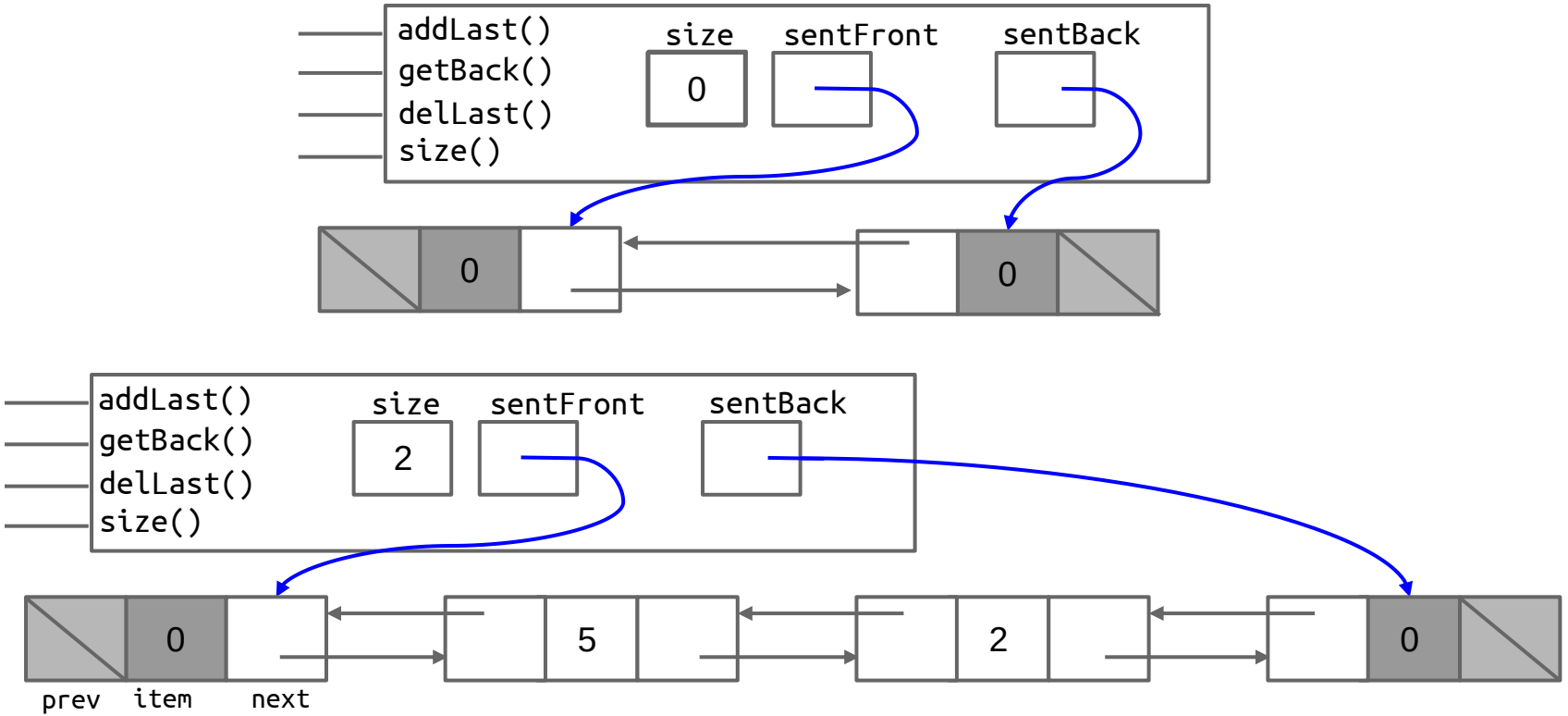
- If you try to code using DLList with just one sentinel,
  there could be a special case where sometimes last point to sentinel or real node



Doubly-Linked List with Double Sentinel

you can use this as an exercise

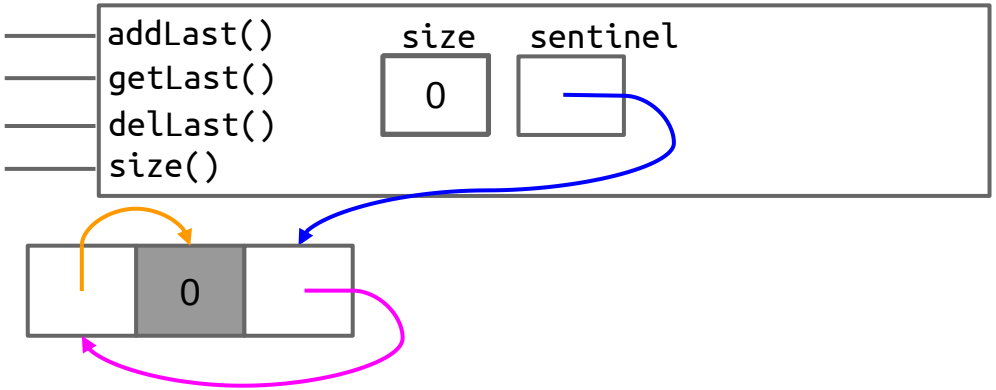
- One possible solution is to have two sentinels:



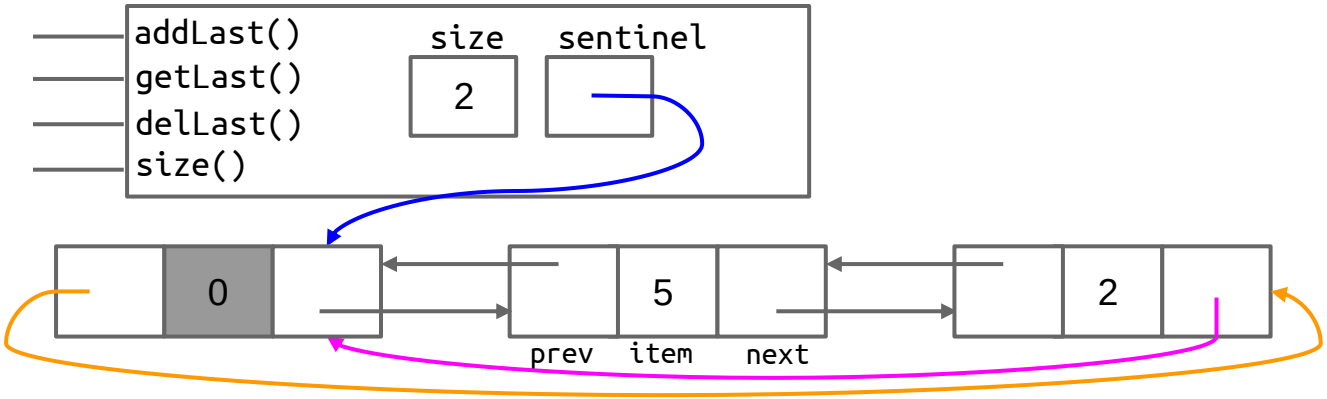
Doubly-Linked List with Circular Sentinel

you will use this for **Lab 5** !

- Another possible solution is to use circular sentinel:



it has **one sentinel** that's both the front and the back



Another Improvement: Generics

- Notice that our list only supports storing an integer
  - it will generate error if we try, for example, to create a list of String
- How do we improve our `SLList` so that it can store any type of item? Use generics!

```
public class SLList4<T>{

    private class Node {
        public T item;
        public Node next;

        public Node(T i, Node n) {
            item = i;
            next = n;
        }
    }
```

usually people like to use T or E here but it can be anything in here

now the class is parameterized by type T, inner class cannot be static

change every type of item in class, from int to T

Using SLList4 with Generics

```
public SLList4() {
    sentinel = new Node( i: null, n: null);
    size = 0;
}

public void addFirst(T item) {
    sentinel.next = new Node(item, sentinel.next);
    size += 1;
}
```

store null as an item in Sentinel node

set all type of item in class to be T

- To use the list, just use it like you use Java's `ArrayList`:

```
public static void main(String[] args) {
    SLList4<String> listStr = new SLList4<>();
    listStr.addFirst( item: "cd");
    listStr.addFirst( item: "ab");
    System.out.println(listStr.getFirst());
}
```

Thank you for your attention !

- In this lecture, you have learned about:
  - Specifications
    - use preconditions and postconditions in method specifications
    - be able to write tests against a specification
  - Linked List 2
    - create and improve your own Linked List
    - equip it with methods that work in constant-time
    - make it able to store items of any data types

- Please continue to Lecture Quiz 5 and Lab 5:
  - to do Lab Exercise 5.1 - 5.4, and
  - to do Exercise 5.1 - 5.4