Advanced Object-Oriented Programming

CPT204 – Lecture 4
Erick Purwanto

Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT204 Advanced Object-Oriented Programming

Lecture 4

Testing 4, Linked List 1

Welcome !

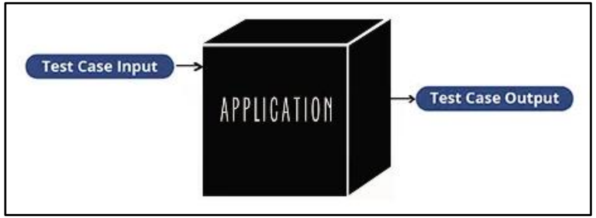
- Welcome to Lecture 4 !

- In this lecture we are going to
  - continue our discussion about Testing
    - how to produce a good test suite
    - when and which part of your code you should test
  - study the mutating or not mutating methods,
    - iteratively or recursively
    - with the running example of our homemade linked list!

Part 1: Testing 4

- We are going to continue to learn testing techniques:
  - Black-box vs White-box Testing
  - Unit vs Integration Testing
  - Automated Regression Testing
  - Testing Documents and Coverage

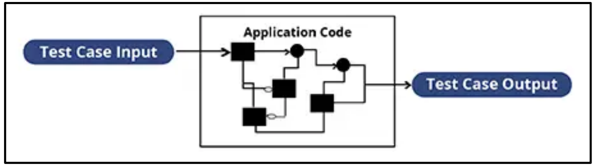
Black-box and White-box Testing (1)



- Recall that the *specification* is the description of the method's behavior — the types of parameters, type of return value, and constraints and relationships between them

- **Black-box testing** means choosing test cases ***only*** from the specification, ***not*** the implementation of the method
  - That's what we've been doing in our lectures and labs so far!
  - We partitioned and looked for corner cases/boundaries in the exercises and assignments, *without* looking at the actual code for these methods

Black-box and White-box Testing (2)



- **White-box testing** (also called *glass box testing*) means choosing test cases with ***knowledge*** of how the method is *actually implemented*
    - For example, if the implementation selects different algorithms depending on the input, then you should partition according to those domains
    - Another example, if you know that the implementation keeps an internal cache that remembers the answers to previous inputs, then you should test repeated inputs

White-box Testing Example

- One more example, for the case of `BigInteger.multiply`,
  - when we finally implemented it, we may have decided to represent small integers with `int` values and large integers with a list of decimal digits
  - this decision introduces new boundary values, presumably at `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, and a new partition around them

White-box Testing Test Cases

- When doing white-box testing, you must take care that your test cases don't require specific implementation behavior that isn't specifically called for by the spec
    - For example, if the spec says “throws an exception if the input is poorly formatted,” then your test *shouldn't* check specifically for a `NullPointerException` just because that's what the current implementation does
    - The specification in this case allows *any* exception to be thrown, so your test case should likewise be general to preserve the implementor's freedom

Documenting Your Testing Strategy (1)

- Document your test strategy at the top of your test class, for example:

```
/*
 * Testing strategy
 *
 * Partition the inputs as follows:
 * text.length(): 0, 1, > 1
 * start:         0, 1, 1 < start < text.length(),
 *                text.length() - 1, text.length()
 * text.length()-start: 0, 1, even > 1, odd > 1
 *
 * Include even- and odd-length reversals because
 * only odd has a middle element that doesn't move.
 *
 * Exhaustive Cartesian coverage of partitions.
 */
```

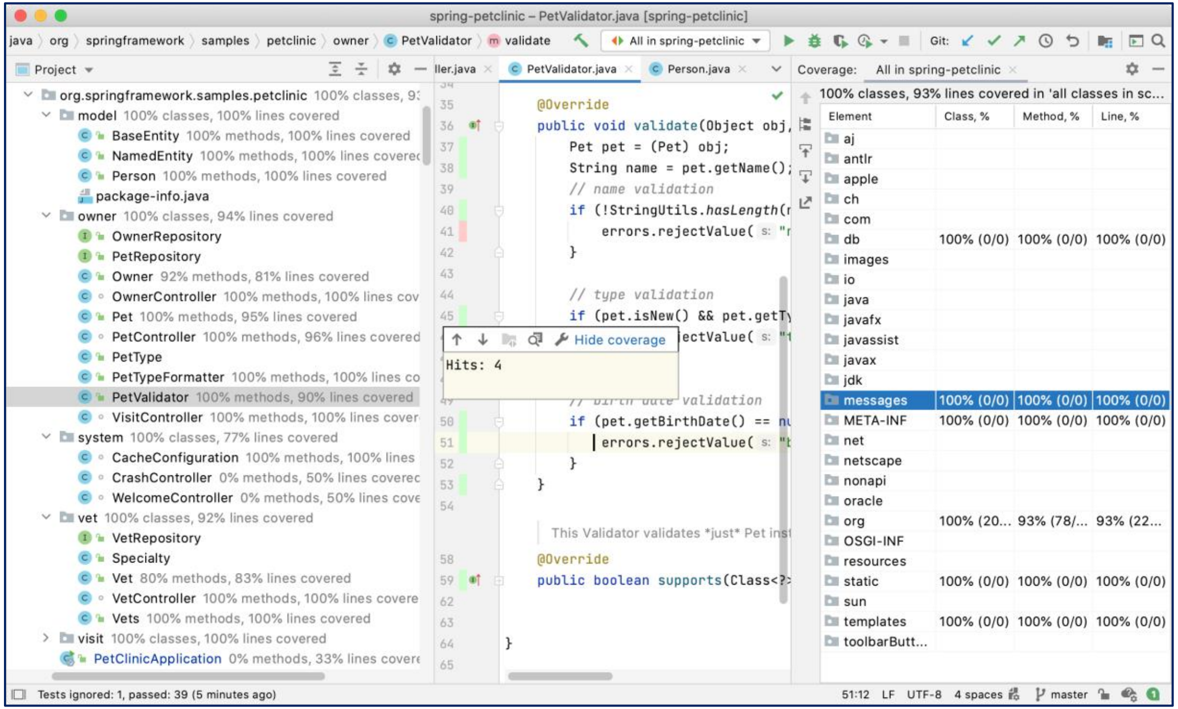
Documenting Your Testing Strategy (2)

- Document how each test case was chosen, including white-box tests:

```
// covers test.length() = 0,
//        start = 0 = text.length(),
//        text.length()-start = 0
@Test public void testEmpty() {
    assertEquals("", reverseEnd("", 0));
}

// ... other test cases ...
```

Coverage (1)

- One way to judge a test suite is to ask how thoroughly it exercises the program
- This notion is called **coverage**



Coverage (2)

- There are three common kinds of coverage:
  - **Statement coverage**: is every statement run by some test case?
  - **Branch coverage**: for every if or while statement in the program, are both the true and the false direction taken by some test case?
  - **Path coverage**: is every possible combination of branches — every path through the program — taken by some test case?
- Branch coverage is stronger (requires more tests to achieve) than statement coverage, and path coverage is stronger than branch coverage
- In industry, 100% statement coverage is a common goal, but even that is rarely achieved due to unreachable defensive code (like “should never get here” assertions)
- However, 100% path coverage is infeasible, requiring exponential-size test suites to achieve

Coverage (3)

- A standard approach to testing is to add tests until the test suite achieves adequate statement coverage, that is, so that every reachable statement in the program is executed by *at least one test case*
- In practice, statement coverage is usually measured by *a code coverage tool*, which counts the number of times each statement is run by your test suite
  With such a tool, white box testing is easy; you just measure the coverage of your black box tests, and add more test cases until all important statements are logged as executed
- You can use a code coverage tool for IntelliJ, for example:
  - IntelliJ IDEA Code Coverage Runner
    https://www.jetbrains.com/help/idea/code-coverage.html
  - EMMA or JaCoCo

Unit Testing and Integration Testing (1)

- A well-tested program will have tests for every individual module (where a module is a method or a class) that it contains
- A test that tests an individual module, in *isolation* if possible, is called **a unit test**
- Testing modules in isolation leads to much easier debugging
  - When a unit test for a module fails,
    you can be more confident that the bug is found *in that module*,
    rather than anywhere in the program

Unit Testing and Integration Testing (2)

- The opposite of a unit test is **an integration test**, which tests a combination of modules, or even the entire program
- If all you have are integration tests, then when a test fails, you have to hunt for the bug
  - it might be anywhere in the program

- Integration tests are still important, because a program can fail at *the connections between modules*
  - for example, one module may be expecting different inputs than it's actually getting from another module
- But if you have a thorough set of unit tests that give you confidence in the correctness of individual modules, then you'll have much less searching to do to find the bug

Unit Testing and Integration Testing Example (1)

- As an example, suppose you're building a web search engine
- Two of your modules are `getWebPage()`, which downloads web pages, and `extractWords()`, which splits a page into its component words:

```java
/** @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) { ... }
```

Unit Testing and Integration Testing Example (2)

- Those two methods are used by another module `makeIndex()` as part of the web crawler that makes the search engine's index:

```java
/** @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) {
    ...
    for (URL url : urls) {
        String page = getWebPage(url);
        List<String> words = extractWords(page);
        ...
    }
    ...
}
```

Unit Testing and Integration Testing Example (3)

- In our test suite, we would want:
  - unit tests just for `getWebPage()` that test it on various URLs
  - unit tests just for `extractWords()` that test it on various strings
  - unit tests for `makeIndex()` that test it on various sets of URLs

Unit Testing and Integration Testing Example (4)

- One mistake that programmers sometimes make is writing test cases for `extractWords()` in such a way that the test cases depend on `getWebPage()` to be correct

- It's better to think about and test `extractWords()` ***in isolation***, and partition it
  - Using test partitions that involve web page content might be reasonable, because that's how `extractWords()` is actually used in the program
  - But don't actually call `getWebPage()` from the test case, because `getWebPage()` may be buggy!
  - Instead, store web page content as a literal string, and pass it directly to `extractWords()`
  - That way you're writing an isolated unit test, and if it fails, you can be more confident that *the bug is in the module it's actually testing*, `extractWords()`

Unit Testing and Integration Testing Example (5)

- Note that the unit tests for `makeIndex()` can't easily be isolated in this way
- When a test case calls `makeIndex()`, it is testing the correctness of ***not only*** the code inside `makeIndex()`, ***but also*** all the methods called by `makeIndex()`
  - If the test fails, the bug might be in any of those methods
  - That's why we want separate tests for `getWebPage()` and `extractWords()`, to increase our confidence in those modules individually and **localize** the problem to the `makeIndex()` code that connects them together

```
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {
    ...
    for (URL url : urls) {
        String page = getWebPage(url);
        List<String> words = extractWords(page);
        ...
```

Unit Testing and Stub

- Isolating a higher-level module like `makeIndex()` is possible if we write **stub versions** of the modules that it calls
  - For example, a stub for `getWebPage()` wouldn't access the internet at all, but instead would return mock web page content no matter what URL was passed to it
- A stub for a class is often called **a mock object**
- Stubs are an important technique when building large systems

Automated Testing

- Nothing makes tests easier to run, and more likely to be run, than complete automation
- **Automated testing** means running the tests and checking their results automatically
  - A test driver should ***not*** be an interactive program that prompts you for inputs and prints out results for you to manually check
  - Instead, a test driver should ***invoke the module itself*** on fixed test cases and automatically check that the results are correct
  - The result of the test driver should be either "all tests OK" or "these tests failed: …"
  - A good testing framework, like **JUnit**, helps you build automated test suites

- Note that automated testing frameworks, like the one we use: **JUnit**, make it easy to run the tests, but *you still have to come up with good test cases yourself*
  - *Automatic test generation* is a hard problem, still a subject of active computer science research

Regression Testing (1)

- Once you have test automation, it's very important to ***rerun your tests when you modify your code***
  - This prevents your program from regressing — introducing other bugs when you fix new bugs or add new features
  - Running *all* your tests after *every* change is called **regression testing**

- Whenever you find and fix a bug, take the input that elicited the bug and add it to your automated test suite as a test case, called a regression test
  - This helps to populate your test suite with good test cases.
  - Remember that a test is good if it elicits a bug — and every regression test did find a bug in one version of your code!
  - Saving regression tests also protects against reversions that reintroduce the bug
  - The bug may be an easy error to make, since it happened once already

Regression Testing (2)

- This idea also leads to *test-first debugging*:
  - When a bug arises, immediately write a test case for it that elicits it, and *immediately add it* to your test suite
  - Once you find and fix the bug, all your test cases will be passing, and you'll be done with debugging and have a regression test for that bug

- In practice, these two ideas, automated testing and regression testing, are almost always used in combination
- Regression testing is only practical if the tests can be run often, automatically
- Conversely, if you already have automated testing in place for your project, then you might as well use it to prevent regressions
- So **automated regression testing** is a *best-practice* of modern software engineering

In-class Quiz

- Which of the following are good times to re-run all your JUnit tests?
- Select one or more:
  - ❑ after rewriting a correct method to make it faster
  - ❑ when using a code coverage tool
  - ❑ after an attempt to fix a bug
  - ❑ before submitting your code to LM Autograder

Part 2: Recursive Linked List

- The codes of the next part of the lecture can be found in **lecture4.zip**
- In addition, we are also going to use the **Java Visualizer** plugin, installed in the first week
    - Read and follow Lab0.pdf if you still have not done so...

My Linked List (1)

- As a running example for our next topics, let us *build a linked list from scratch*
  - you can find the lecture codes in LMO

MyList1.java

```java
public class MyList1 {
    private int value;
    private MyList1 next;

    public static void main(String[] args) {
        MyList1 list = new MyList1();
        list.value = 5;
        list.next = null;

        list.next = new MyList1();
        list.next.value = 2;

        list.next.next = new MyList1();
        list.next.next.value = 10;
    }
}
```

we start with the simplest one `MyList1` has only 2 member variables / instance variables

an integer value, which is the data we wish to store inside

and a reference to another `MyList1` object (people also call this pointer or address to `MyList1` object)

set them to be private so they can only be accessed and modified from inside class

My Linked List (2)

- As a running example for our next topics, let us *build a linked list from scratch*
  - you can find the lecture codes in LMO

MyList1.java

```java
public class MyList1 {
    private int value;
    private MyList1 next;

    public static void main(String[] args) {
        MyList1 list = new MyList1();
        list.value = 5;
        list.next = null;

        list.next = new MyList1();
        list.next.value = 2;

        list.next.next = new MyList1();
        list.next.next.value = 10;
    }
}
```

turns out those two are enough to implement a linked list!

first, we create our list, set its value, set reference to null since no other object follows

if we want to store another int, we create a new object, and store its reference in list.next

and so on …
can we do better than writing list.next.next.next ?

Java Visualizer (1)

- But first, let me introduce you to a way to help us understand what happened

MyList1.java

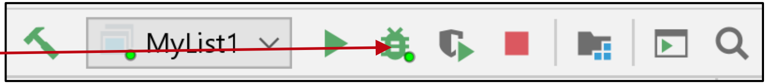
```java
public class MyList1 {
    private int value;
    private MyList1 next;

    public static void main(String[] args) {
        MyList1 list = new MyList1();
        list.value = 5;
        list.next = null;

        list.next = new MyList1();
        list.next.value = 2;

        list.next.next = new MyList1();
        list.next.next.value = 10;
    }
}
```

first, run the code
you notice that nothing happens

next, set a breakpoint
by clicking on this area
a red circle will appear

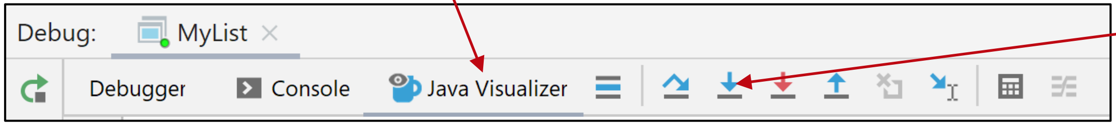
Java Visualizer (2)

then click Debug
on your upper right corner



then click Java Visualizer
below to the left

and click Step Into
to execute your code
line-by-line



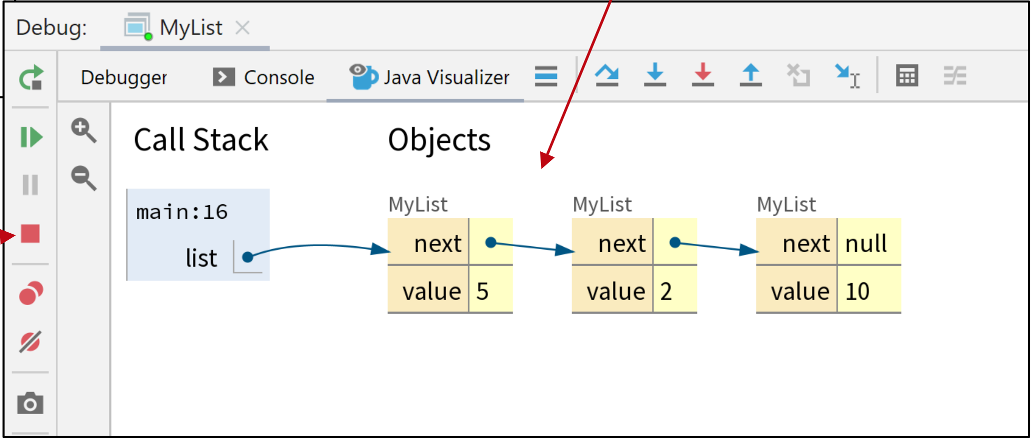
Java Visualizer (3)

MyList1.java

```java
public class MyList1 {
    private int value;
    private MyList1 next;

    public static void main(String[] args) {  args: {}
        MyList1 list = new MyList1();  list: MyList1@790
        list.value = 5;
        list.next = null;

        list.next = new MyList1();
        list.next.value = 2;  list: MyList1@790

        list.next.next = new MyList1();
        list.next.next.value = 10;
    }
}
```

this blue line indicates the next line to be executed when you click Step Into

you will see **a visualization** of your linked list !



click this red box to stop the debugger and then continue coding

My Linked List 2 with Constructor

- Here's our second try, this time with a constructor

MyList1.java × MyList2.java ×

```java
public class MyList2 {
    private int value;
    private MyList2 next;

    public MyList2(int value, MyList2 next) {
        this.value = value;
        this.next = next;
    }

    public static void main(String[] args) {
        MyList2 list = new MyList2( value: 10, next: null);
        list = new MyList2( value: 2, list);
        list = new MyList2( value: 5, list);
    }
}
```

you have learned about constructor in CSE105, it initializes the instance variables

we build the same list as before, **from the end**

each time we update the reference to next to list

and then we update the reference to list

visualize it, change to `MyList2` before Debug

My Linked List 3 with Size: Iterative

- Next, we will equip our linked list with size and get methods

```java
/**
 * @return the size of the MyList iteratively.
 */
public int iterSize() {
    MyList3 p = this;
    int size = 0;
    while (p != null) {
        size += 1;
        p = p.next;
    }
    return size;
}
```

let's first do this using iteration

we use pointer p to the current MyList3 object

we keep looping, moving the pointer, and increment size until we point to null

stop! can you think first,
how to implement size *recursively*?
what is the base case?
what is the recursive step?

My Linked List 3 with Size: Recursive

- Next, we will equip our linked list with size and get methods

```java
/**
 * @return the size of the MyList recursively.
 */
public int recSize() {
    // base case
    if (next == null) {
        return 1;
    }
    // recursive step
    return 1 + this.next.recSize();
}
```

now let's do this using recursion

when next is null, our list only stores one value

otherwise, given the size of the next part of the list, we add 1 to it

you can always use the visualizer and running the code step-by-step to help you understand what's going on

My Linked List 3 with Get: Recursive

- Next, we will equip our linked list with size and get methods

```java
/**
 * @param i is a valid index of MyList.
 * @return the ith value of this MyList.
 */
public int get(int i) {
    // base case
    if (i == 0) {
        return value;
    }
    // recursive step
    return next.get(i - 1);
}
```

we implement get recursively, to return the ith value of list, assume it exists

if value can be found here, (at index 0), return it

otherwise, get the (i-1)th value from the remaining list

as an exercise, you try implement get *iteratively*. again, use visualizer to help you understand this.

My Linked List 3 Additional Methods: ofEntries

- We also equip it with two additional methods for ***testing purposes***
  - we will cover this in future lectures, you ***don't*** have to understand it now

```java
/**
 * @param args is a variable number of integers.
 * @return a new MyList containing the integers in args.
 */
public static MyList3 ofEntries(Integer... args) {
    MyList3 result, p;
    if (args.length > 0) {
        result = new MyList3(args[0], null);
    } else {
        return null;
    }
    int k;
    for (k = 1, p = result; k < args.length; k += 1, p = p.next) {
        p.next = new MyList3(args[k], null);
    }
    return result;
}
```

this method is used to create `MyList3` objects by specifying the values as sequence of integers

for example, to create [1, 2, 3],
`MyList3 list = MyList3.ofEntries(1, 2, 3);`

My Linked List 3 Additional Methods: equals

- We also equip it with two additional methods for ***testing purposes***
  - we will cover this in future lectures, you ***don't*** have to understand it now

```java
/**
 * @param l is a MyList object.
 * @return true iff l is a MyList object containing the same sequence of
 * integers as this.
 */
public boolean equals(Object l) {
    if (!(l instanceof MyList3)) {
        return false;
    }
    MyList3 list = (MyList3) l;
    MyList3 p;
    for (p = this; p != null && list != null; p = p.next, list = list.next) {
        if (p.value != list.value) {
            return false;
        }
    }
    if (p != null || list != null) {
        return false;
    }
    return true;
}
```

this method is used to compare two `MyList3` objects, will return true iff their values are the same

for example, `list1.equals(list2)` is true iff `list1` and `list2` are `MyList3` objects containing same integers

it is used by `assertEquals()` to compare two `MyList3` objects

Thank you for your attention !

- In this lecture, you have learned:
  - Testing 4
    - White Box vs Black Box, Unit vs Integration, Automated Regression
    - Testing Documents and Coverage
  - Linked List 1
    - Creating your own Linked List
    - Equipping it with methods iteratively and recursively

- Please do Lecture Quiz 4, and continue to Lab 4:
  - to add more methods to the linked list *iteratively* and *recursively*,
  - to do Lab Exercise 4.1 - 4.4, and
  - to do Exercise 4.1 - 4.4

`MyList3` is now `MyList` in Lab 4