FREE SAMPLE

- Get That Job!
- Become The Best Candidate
- Increase Your Earnings

TOP 30 JAVA™ INTERVIEW CODING TASKS

WITH WINNING SOLUTIONS

Matthew Urban

TOP 30 JAVA™ INTERVIEW CODING TASKS

WITH WINNING SOLUTIONS

TOP 30 Java Interview Coding Tasks with Winning Solutions
by Matthew Urban

Copyright © 2018 net-boss. All rights reserved.
Published by net-boss https://net-boss.org

July 2018: First Edition

Proofreader: Alex Solsbery
Illustrator: Matthew Urban
Cover designer: Karolina Kaiser
Composition: Karolina Kaiser (mobisfera.pl)

ISBN 978-83-65477-10-1

ISBN 978-83-65477-10-1
9 788365 477101

Preface

This book contains the 30 most popular coding tasks used by Java developers during a job interview. There is no need to waste time to get to know all possible coding tasks, which recruiters will not ask you to solve. Instead, familiarize yourself with the most frequent ones used and prepare for your job interview in the most effective way possible.

Matthew Urban
IT specialist, Senior Java developer

1. Reverse a text.

From my experience, reverse a text is a very popular coding task used during job interview. Text manipulation methods are the ones mostly used by programmers. Although you do not need to implement them, it is desired to understand how such operations are performed under the hood. The low-level details summed together have a significant impact on overall system performance.

Solution

The `String` class represents a text value in Java which is built from an array of characters. To implement a custom method to reverse a `String` you also have to operate on an array of characters. There are many solutions to reverse a `String`, but the most optimal should not allocate additional memory if there is no need. It is recommended to take an input `char[]` array, iterate through it and switch the first element with the last, the second element with the penultimate, etc. until half of the array is reached as presented in Listing 1.1.

Listing 1.1 – Reverse a text algorithm.

```java
public class StringUtils {
   public static String reverse(String input) {
      if (input == null) {
         return "";
      }
      char[] array = input.toCharArray();
      final int halfLength = array.length/2;
      int idx2;
      char clipboard;
      for (int idx1=0; idx1 < halfLength; idx1++) {
         idx2 = array.length - 1 - idx1;
         clipboard = array[idx1];
         array[idx1] = array[idx2];
         array[idx2] = clipboard;
      }
      return String.valueOf(array);
   }
}
```

Tests

The `StringUtils.reverse()` method is a good example of code which is easy to test with unit tests. Listing 1.2 presents an example implementation of such a unit test. In this case, a test is parametrized with two variables: `expected` and `input` values. The `data()` method, is a factory method which returns all combinations of input and expected data that need to be tested. Remember, preparing test cases and a method body simultaneously (TDD) results in high-quality code.

Listing 1.2 – Unit test used to verify the `StringUtils.reverse()` method.

```
@RunWith(Parameterized.class)
public class StringUtilsReverseTest {
   @Parameters(name = "({index}): reverse({1})={0}")
   public static Collection data() {
      return Arrays.asList(new Object[][]{
         {"", null},
         {"", ""},
         {"a", "a"},
         {"1234567890", "0987654321"},
         {"aBbA", "AbBa"}
      });
   }

   @Parameter(value = 0)
   public String expected;

   @Parameter(value = 1)
   public String input;

   @Test
   public void reverse() {
      assertEquals(expected, StringUtils.reverse(input));
   }
}
```

2. Design and implement LRU cache.

The main idea behind the LRU (Least Recently Used) cache is to "promote" elements which are most frequently used. The most recently used elements stay in cache, while least used entries are removed from the cache automatically. Typically, LRU cache is implemented by using a doubly linked list and hash map data structure. A combination of both structures allows fastest retrieval time (by hash map) and maintaining their access-order (by linked list).

Solution

Many developers start to write their own LRU cache from scratch. This is not the best approach, because Java API already provides a ready to use `LinkedHashMap` class, which is well-suited to build LRU cache. When the recruiter sees that the developer reuses code, instead of writing the entire solution from scratch, he can be sure that he has found an experienced programmer. The benefits of reusing code are: new code is created faster; the cost of maintenance is lower and there is a less risk of unpredictable behavior of the program. Listing 2.1 presents an example implementation of LRU cache, which reuses the `LinkedHashMap` class. The default ordering of elements in `LinkedHashMap` class is insertion-order, but you can indicate in the constructor that you prefer access-order. If you would like to limit the number of elements in a cache (set max cache size) you need to override the `removeEldestEntry()` method, which was designed for that purpose.

Listing 2.1 – Example implementation of LRU cache.

```java
public class LRUCache<K,V> {
   private Map<K,V> map;

   public LRUCache(int cacheSize) {
      map = new LinkedHashMap<K,V>(16, 0.75f, true) {
         @Override
         protected boolean removeEldestEntry(Map.Entry eldest) {
            return size() > cacheSize;
         }
      };
   }
```

```
    public V get(K key) {
        return map.get(key);
    }

    public void set(K key, V value) {
        map.put(key, value);
    }
}
```

Tests

To implement a unit test which verifies the correctness of LRU cache eviction policy, you need to be sure which element and why should be removed. It is helpful to print the content of cache when tests are created. When you are sure that your tests are correct, just remove the `System.out.println()` invocations. Listing 2.2 presents a simple unit test which can be implemented during the interview. Before such solution can be used in production environment, you need to extend the unit test with more sophisticated use cases.

Listing 2.2 – Unit test used to verify the `LRUCache` class.

```
public class LRUCacheTest {
    LRUCache<Integer, Integer> cache = new LRUCache<>(4);

    @Test
    public void evictLeastRecentlyUsed() {
        cache.set(1, 11);
        cache.set(2, 22);
        cache.set(3, 33);
        cache.set(4, 44);
        cache.set(5, 55);

        assertNull(cache.get(1));
        assertEquals(22, cache.get(2).intValue());
        assertEquals(33, cache.get(3).intValue());
        assertEquals(44, cache.get(4).intValue());
        assertEquals(55, cache.get(5).intValue());

        cache.set(6, 66);

        assertNull(cache.get(1));
        assertNull(cache.get(2));
        assertEquals(33, cache.get(3).intValue());
        assertEquals(44, cache.get(4).intValue());
        assertEquals(55, cache.get(5).intValue());
        assertEquals(66, cache.get(6).intValue());
        cache.set(7, 77);

        assertNull(cache.get(1));
        assertNull(cache.get(2));
        assertNull(cache.get(3));
        assertEquals(44, cache.get(4).intValue());
        assertEquals(55, cache.get(5).intValue());
        assertEquals(66, cache.get(6).intValue());
        assertEquals(77, cache.get(7).intValue());

        cache.set(8, 88);

        assertNull(cache.get(1));
        assertNull(cache.get(2));
        assertNull(cache.get(3));
        assertNull(cache.get(4));
        assertEquals(55, cache.get(5).intValue());
        assertEquals(66, cache.get(6).intValue());
        assertEquals(77, cache.get(7).intValue());
        assertEquals(88, cache.get(8).intValue());
    }
}
```

3. Compute the distance between two points in 3D space.

The distance between two points, also known as Euclidean distance is defined by the following formula: *sqrt*$((x-y)^2)$. Depending on the number of dimensions, the distance between two points is defined by different equations. For example, on a 2D surface where points have two coordinates, the distance between them is: *sqrt*$((x_1-y_1)^2+(x_2-y_2)^2)$. Your task is to prepare a formula and implement a function which calculates the distance between two points in 3D space.

Solution

The 3D surface differs from 2D in that it has one additional dimension: height. The equation which measures distance between two points must only include the third coordinate: *sqrt*$((x_1-y_1)^2+(x_2-y_2)^2+(x_3-y_3)^2)$. Having such an equation you need to write it down using Java language. Listing 3.1 presents an example of such function. Please notice that `Math` class provides a ready to use square root function.

Listing 3.1 – Distance between two points in 3D space.

```
public class Point {
   final double x;
   final double y;
   final double z;

   public Point(double x, double y, double z) {
      this.x = x;
      this.y = y;
      this.z = z;
   }

   double distance(Point p2) {
      double dx = this.x - p2.x;
      double dy = this.y - p2.y;
      double dz = this.z - p2.z;
```

```
        return Math.sqrt(dx*dx + dy*dy + dz*dz);
    }
}
```

Tests

To test if your function returns expected results, calculate the distance between points using the calculator, and then verify those results inside your unit test.

Listing 3.2 – A unit test used to verify `Point.distance()` method.

```
public class PointDistanceTest {
    @Test
    public void distance() {
        Point p1 = new Point(8, 2, 6);
        Point p2 = new Point(8, 6, 3);
        assertTrue(Double.compare(5, p1.distance(p2)) == 0);
        assertTrue(Double.compare(5, p2.distance(p1)) == 0);
    }
}
```

4. Compare application version numbers.

An application version number is used to “tag” the released version of an application. In most cases, it is a number created in the form of decimal numbers and dots. The numbers at the left side are more valuable than numbers on the right side of the dot. For example:

- application version number `15.1.1` (and `15.1`) is greater than `14.13.10` because `15` > `14`,
- application version number `14.13.10` is greater than `14.10.55` because `13` > `10`,
- application version number `14.10.55` is greater than `14.10.20` because `55` > `23`.

Solution

The first idea which comes to your mind may be to remove the dots and compare the numbers. Unfortunately, such comparator would return incorrect results for versions such as `15.1` and `14.1.2` because `151 < 1412`. The correct solution must split the numbers and compare each from the left to the right side. Listing 4.1 presents an example implementation of `VersionNumberComparator` class.

Listing 4.1 – Compare application version numbers.

```
public class VersionNumberComparator implements Comparator<String> {
   @Override
   public int compare(String version1, String version2) {
      Integer[] array1 = Arrays.stream(version1.split("\\."))
              .map(Integer::parseInt)
              .toArray(Integer[]::new);
      Integer[] array2 = Arrays.stream(version2.split("\\."))
              .map(Integer::parseInt)
              .toArray(Integer[]::new);
      int length1 = array1.length;
      int length2 = array2.length;
      int idx = 0;
      while (idx < length1 || idx < length2) {
         if (idx < length1 && idx < length2) {
            if (array1[idx] < array2[idx]) {
                    return -1;
                } else if (array1[idx] > array2[idx]) {
                    return 1;
                }
            } else if (idx < length1) {
                if (array1[idx] != 0) {
                    return 1;
                }
            } else if (idx < length2) {
                if (array2[idx] != 0) {
                    return -1;
                }
            }
            idx++;
        }
        return 0;
    }
}
```

Tests

During the coding interview, in most cases you are under time-pressure. Do not waste your time to prepare the best unit test in the world, instead focus on most important test cases, like those which are presented in Listing 4.2.

Listing 4.2 – A unit test used to verify `VersionNumberComparator` function.

```
public class VersionNumberComparatorTest {
    Comparator<String> vnc = new VersionNumberComparator();

    @Test
    public void compareVersions() {
        assertTrue(vnc.compare("14", "14.0") == 0);
        assertTrue(vnc.compare("15", "14") > 0);
        assertTrue(vnc.compare("15.1", "14.13.10") > 0);
        assertTrue(vnc.compare("15.1", "15.1.0") == 0);
        assertTrue(vnc.compare("15.1.1", "14.13.10") > 0);
        assertTrue(vnc.compare("14.13", "14.10.55") > 0);
        assertTrue(vnc.compare("14.13.10", "14.10.55") > 0);
        assertTrue(vnc.compare("14.10.55", "14.10.20") > 0);
        assertTrue(vnc.compare("14.10.20", "14.10.20") == 0);
    }
}
```

5. Reverse a linked list.

First, let's note the basic concept behind a linked list data structure. A linked list is a data structure which contains a chain of nodes. Each node stores a reference to the next node, which allows a linked list to grow dynamically and save as many elements as needed. Listing 5.1 presents an example implementation of a linked list data structure in Java.

Listing 5.1 – Example implementation of linked list data structure.

```
class LinkedList<T> {
   Node head;

   private class Node {
      final T value;
      Node next;

      Node(T value, Node next) {
         this.value = value;
         this.next = next;
      }
   }
}
```

The `LinkedList` class contains a `head` field which is the first node of the linked list. Each node implemented by `Node` class, contains a `next` field, which points to the next element in a sequence. In such way, the head element points to next element, the next element to another, and at the end, the last element points to `null` value. Before you implement a reverse algorithm for a linked list, you need to implement methods which allow you to put elements inside and verify the presence and order of those values. Listing 5.2 presents example implementations of `add()` and `toString()` methods.

Listing 5.2 – Example implementation of `add()` and `toString()` methods.

```
class LinkedList<T> {
   Node head;

   public void add(T value) {
      Node node = new Node(value, null);
      if (head == null) {
            head = node;
        } else {
            Node last = head;
            while (last.next != null) {
                last = last.next;
            }
            last.next = node;
        }
    }

    @Override
    public String toString() {
        StringJoiner joiner = new StringJoiner(" -> ", "[", "]");
        Node last = head;
        while (last != null) {
            joiner.add(last.value.toString());
            last = last.next;
        }
        return joiner.toString();
    }

    private class Node {
        final T value;
        Node next;

        Node(T value, Node next) {
            this.value = value;
            this.next = next;
        }
    }
}
```

Next, you should verify if your first implementation of `LinkedList` class works as expected. Listing 5.3 presents a simple client code, which verifies the basic functionality of your own `LinkedList` class.

Listing 5.3 – Example usage of `LinkedList` class.

```
LinkedList<String> list = new LinkedList<>();
list.add("a1");
list.add("a2");
list.add("a3");
list.add("a4");
list.add("a5");

//prints [a1 -> a2 -> a3 -> a4 -> a5]
System.out.println(list);
```

Solution

There are many algorithms to reverse a linked list. The most popular, and simultaneously most expected solution during a coding interview is the simplest: the switch-three-references approach. The concept behind it is to traverse through a linked list from head to the last element and move elements by switching references with the help of temporary variables. Listing 5.4 presents a correct implementation of the switch-three-references approach.

Listing 5.4 – Reverse a linked list algorithm.

```java
class LinkedList<T> {
   Node head;

   //...

   public void reverse() {
      if (head == null) {
         return;
      }
      Node p1 = head;
      Node p2;
      while (p1.next != null) {
         p2 = p1.next;
         p1.next = p2.next;
         p2.next = head;
         head = p2;
      }
   }
}
```

Sometimes it can be difficult to find a correct solution during the interview, especially when a developer is stressed. That is why it is recommended to become familiar with the main idea beforehand. In each loop cycle, the next element is moved to the beginning of the list by switching the references. The procedure is repeated until the end of the list is reached. Listing 5.5 presents an example list after each loop cycle.

Listing 5.5 – Phases of reverse a linked list algorithm.

```
[a1 -> a2 -> a3 -> a4 -> a5]

(move the a2 to the beginning)
[a2 -> a1 -> a3 -> a4 -> a5]
```

```
(move the a3 to the beginning)
[a3 -> a2 -> a1 -> a4 -> a5]

(move the a4 to the beginning)
[a4 -> a3 -> a2 -> a1 -> a5]

(move the a5 to the beginning)
[a5 -> a4 -> a3 -> a2 -> a1]
```

To be able to switch elements, three references are used: `p1`, `p2` and `head`. Reference `p1` is used to store the `head` node, which after the reverse becomes the last node. The `p2` is used to traverse through a list and move next elements in the sequence (second, third, etc..) from the middle to the beginning of the list. The `head` reference is used to save a new first element of the list for each loop cycle. For example, in the first loop cycle, the `p2` variable is used to point to the second element, meaning the element where `p1.next` points to. Next, references are switched, so the first element points to the third element, the second element points to the first, and the `head` becomes the second element as presented in Figure 5.1.

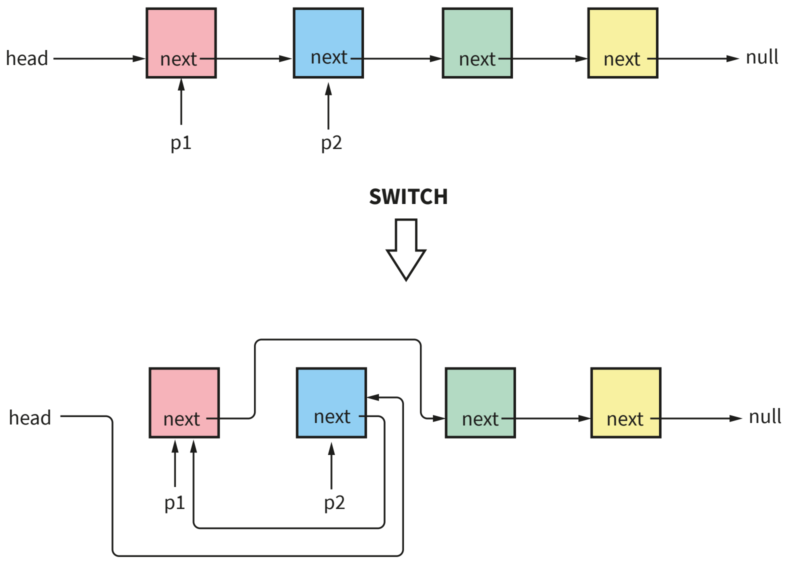

Figure 5.1 – To reverse a linked list, three references are switched.

Tests

It is recommended to implement such a method using the TDD approach. This helps you to create a correct solution step by step. Listing 5.6 presents an example implementation of a unit test.

Listing 5.6 – Unit tests used to verify the reverse algorithm.

```java
public class ReverseLinkedListTest {
    @Test
    public void reverseEmptyList() {
        LinkedList<String> list = new LinkedList<>();
        list.reverse();
        assertEquals("[]", list.toString());
    }

    @Test
    public void reverseOneElementList() {
        LinkedList<String> list = new LinkedList<>();
        list.add("a1");
        list.reverse();
        assertEquals("[a1]", list.toString());
    }

    @Test
    public void reverseManyElementsList() {
        LinkedList<String> list = new LinkedList<>();
        list.add("a1");
        list.add("a2");
        list.add("a3");
        list.add("a4");
        list.add("a5");
        assertEquals("[a1 -> a2 -> a3 -> a4 -> a5]", list.toString());
        list.reverse();
        assertEquals("[a5 -> a4 -> a3 -> a2 -> a1]", list.toString());
    }
}
```