Q1

Annotation:

@Test annotation specifies that method is the test method.

@Test(timeout=1000) annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

@BeforeClass annotation specifies that method will be invoked only once, before starting all the tests.

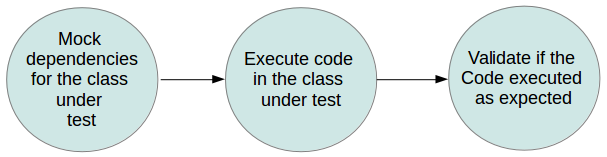
@Before annotation specifies that method will be invoked before each test. @After annotation specifies that method will be invoked after each test. @AfterClass annotation specifies that method will be invoked only once, after finishing all the tests

Q2

1. Mock away external dependencies and insert the mocks into the code under test

2. Execute the code under test

3. Validate that the code executed correctly



Mockito provides several methods to create mock objects:

1. Using the static mock() method.

2. Using the @Mock annotation.

If you use the @Mock annotation, you must trigger the creation of annotated objects. The MockitoRule allows this. It invokes the static method MockitoAnnotations.initMocks(this) to populate the annotated fields. Alternatively you can use @RunWith(MockitoJUnitRunner.class).

Q3

We need to use mock to create a mock instance of the service layer, and get the "service layer message received" as a message when calling getMessage () the mock object, to ensure that getMessage () is called at least once. Finally run the test and verify whether it is successful.

Q4

Assert class have multiple methods to test the code. The methods can be used in @Test.

The code is shown in the Java file.

Q5

The sub-threads will stop immediately when the main thread finished executing. To make sure the sub-threads work correctly, thread.join() can be added or Thread.sleep(); can be added to ensure that the main thread finishes after the sub-threads finishes.

The code is shown in the Java file.

Q6

System.gc() and System.runFinalization() can be used to ensure the gc works correctly.

Use Runtime.getRuntime().freememory() to read the free memory before and after the methods that may cause memory leak.

Compare the free memories using assertEquals. If they are the same, no memory leak detected. If they are different, memory leak is detected.

The code is shown in the Java file.

Q7

The push () method observes the current top node, builds a new node and puts it on the stack, and then, if the top node does not change after the initial observation, then installs the new node. If the CAS fails, meaning that another thread has modified the stack, the process will restart.

public class ConcurrentStack<E> {

    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

    public void push(E item) {

        Node<E> newHead = new Node<E>(item);

        Node<E> oldHead;

        do {

            oldHead = head.get();

            newHead.next = oldHead;

        } while (!head.compareAndSet(oldHead, newHead));

    }

    public E pop() {

        Node<E> oldHead;

        Node<E> newHead;

        do {

            oldHead = head.get();

            if (oldHead == null)

                return null;

            newHead = oldHead.next;

        } while (!head.compareAndSet(oldHead,newHead));

        return oldHead.item;

    }

    static class Node<E> {

        final E item;

        Node<E> next;

        public Node(E item) { this.item = item; }

    }

}

Q8

Inserting an element at the end of a linked list usually involves updating two pointers: the "tail" pointer always points to the last element in the list, and the "next" pointer points to the newly inserted element from the last element in the past. Because two pointers need to be updated, two CASs are required.

Link from the current last node of the queue to the new node, and move the tail pointer to the new last node. If the first step fails, the state of the queue remains unchanged and the insert thread will continue to retry until it succeeds. Once the operation is successful, the insertion is considered effective, and other threads can see the modification.

public class LinkedQueue <E> {

    private static class Node <E> {

        final E item;

        final AtomicReference<Node<E>> next;

        Node(E item, Node<E> next) {

            this.item = item;

            this.next = new AtomicReference<Node<E>>(next);

        }

    }

    private AtomicReference<Node<E>> head

        = new AtomicReference<Node<E>>(new Node<E>(null, null));

    private AtomicReference<Node<E>> tail = head;

    public boolean put(E item) {

        Node<E> newNode = new Node<E>(item, null);

        while (true) {

            Node<E> curTail = tail.get();

            Node<E> residue = curTail.next.get();

            if (curTail == tail.get()) {

                if (residue == null) {

                    if (curTail.next.compareAndSet(null, newNode)) {

                        tail.compareAndSet(curTail, newNode);

                        return true;

                    }

                } else {

                    tail.compareAndSet(curTail, residue);

                }

            }

        }

    }

}