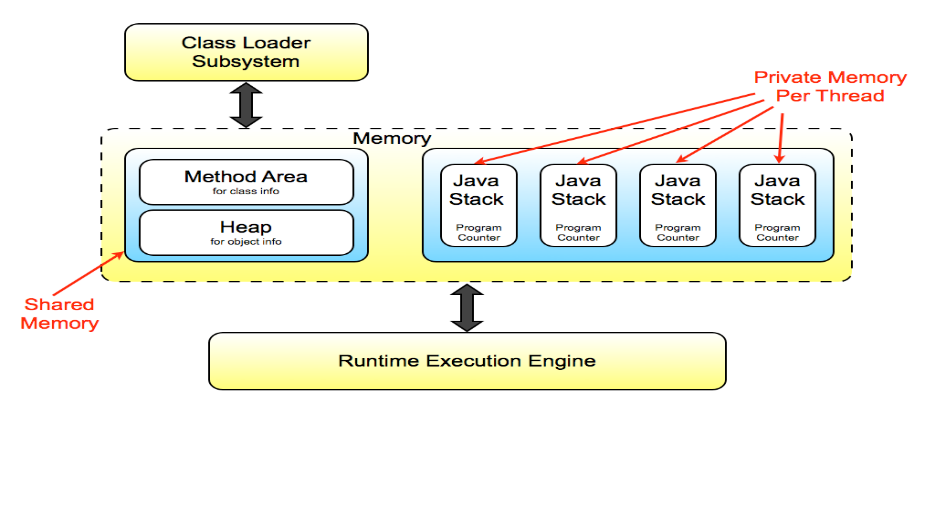
**QUESTION 1**

1. **COMP1206**: Which one of the following statements about Java programming is **false**?

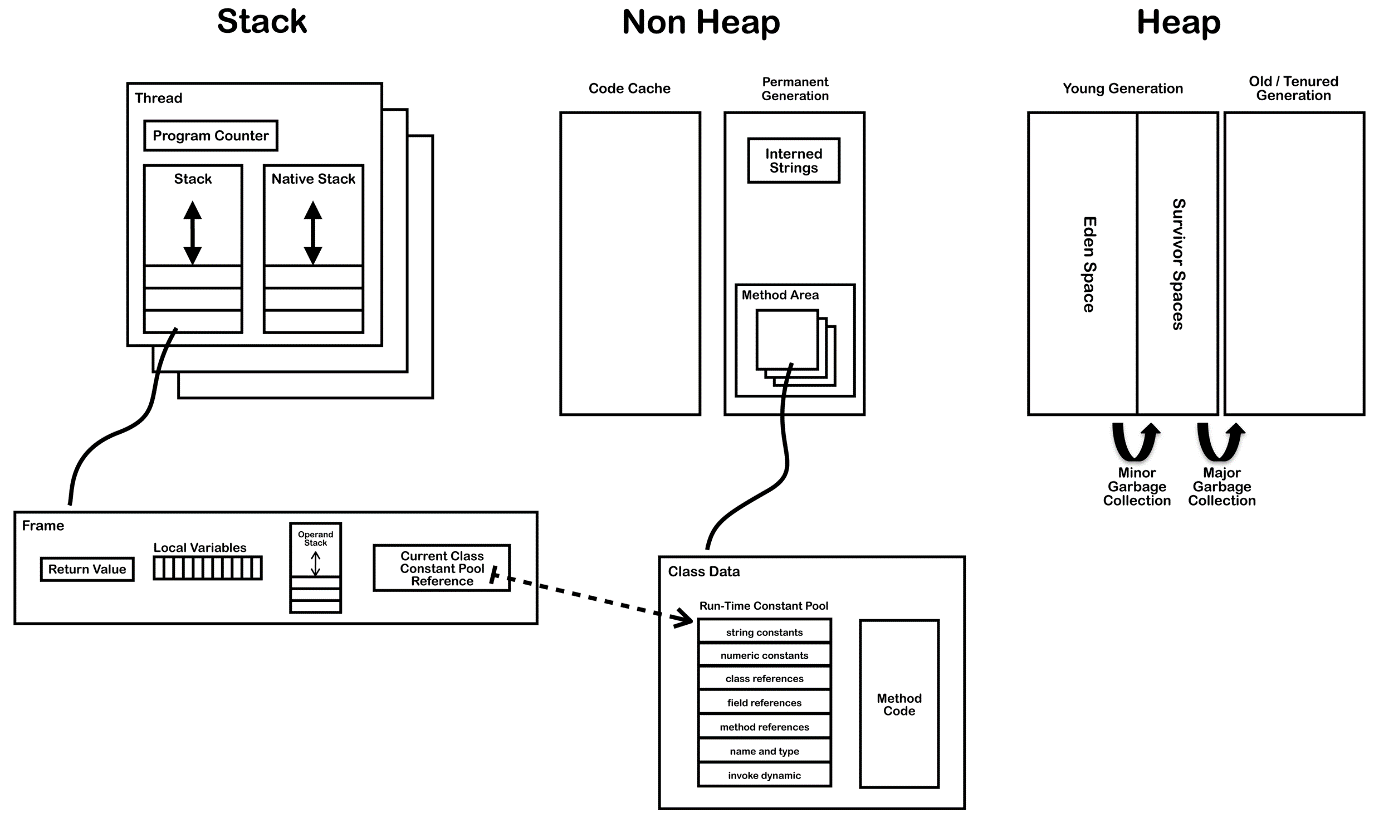
*(select one only)*

|  |  |  |
| --- | --- | --- |
|  |  | The heap is used for allocating class instances and arrays at run-time. |
|  |  | *This is true. The heap is shared between all threads in a JVM.* ***Object instances are stored in the heap.*** *Java**threats* ***arrays*** *as* ***object instances*** *and as such they are also* ***stored******in******the******heap****.*  *(lecture 12)* |
|  |  | Each instance of the same string shares the same immutable interned string.  *This is false.* ***Immutable*** *String means that it is a* ***constant*** *String. (All String are immutable by default). To* ***intern******a******String*** *means that when* ***multiple******Strings*** *with the \*****same****\** ***value*** *are created, they are* ***stored*** *in* ***one******place*** *to preserve memory. With Java,* ***all******constant******Strings******are******interned****. But that’s only constant String expressions, and only when they’re compiled at the same time. Similarly, dynamically created Java String (e.g., the output of various toString() methods) won’t be interned unless the method specifically requests it via String.intern().* |
|  |  | The stack stores values of primitive type and references to objects / arrays.  *This is true. When a thread calls a method, a new stack frame is pushed on to the thread’s Java Stack. All local variables to the method, parameter to the method and temporary calculation is done within this frame. (lecture 12)* |
|  |  | The runtime constant pool for a class contains the bytecode for each of its methods.  *This is true. A class file keeps all its symbolic references in one place – the constant pool. Each class file has a constant pool, and each class or interface loaded by the JVM has an internal version of its constant pool called the runtime constant pool. The runtime constant pool is an implementation-specific data structure that maps to the constant pool in the class file. Thus, after a type is initially loaded, all the symbolic references from the type reside in the type’s runtime constant pool.*  *TLDR there is a difference between constant pool and runtime constant pool but that is not the point here****. The constant pool is part of the Method Area. Imagine it as a big array of all the entities that are referred to in the class definition for the type. These “entities” can be Strings, integers, types, fields, methods etc****. (lecture 12)* |
|  |  |  |

Notes:



* The **memory** of the **JVM** is organised into **shared** and **non-shared** data areas.
* There is **one** **heap** and **one** **method** area **per** **JVM** (**shared**).
* Each **thread** has its own Java Stack and Program counter (**non-shared**).
* There are **no** **registers** for **temporary** **storage**.
* JVM has a **Stack Based** architecture.
* The **Program** **Counter** holds the **address** **of** the **next** **instruction** to execute for each thread.
* The **Java** **Stack** holds **state** **information** for that **thread**. (which methods have been called). It is also used for **storing** **local** **variables**, **parameters** and **performing** **calculations**.



The frame is where the operands (operation instructions) reside and that is where the dynamic linking occurs. Using the constant pool, we keep track of the class and its members. Each frame contains a reference to the runtime constant pool. The reference points to the constant pool for the class of the method being executed in that frame.

When a Java file is compiled, all references to variables and methods are stored in the class’s constant pool as a symbolic reference. A symbolic reference is a logical reference not a reference that points to a physical memory location.

|  |  |  |
| --- | --- | --- |
|  |  |  |

**QUESTION 2**

1. **COMP1206:** In a C program, suppose that POINT is a pointer variable of type int, VAR is an integer variable and ARRAY is an array of integers. Both VAR and ARRAY have been declared and initialised. Which of the following statements are correct ways to initialise POINT?

*(select all that apply)*

|  |  |  |
| --- | --- | --- |
|  |  | POINT = VAR; |
|  |  | POINT = &VAR; |
|  |  | POINT = \*VAR; |
|  |  | \*POINT = VAR; |
|  |  | \*POINT = &VAR; |
|  |  | \*POINT = &ARRAY; |
|  |  | POINT = &ARRAY; |
|  |  | &POINT = ARRAY; |
|  |  | POINT = ARRAY; |
|  |  | POINT = \*ARRAY; |

Pointers in C:

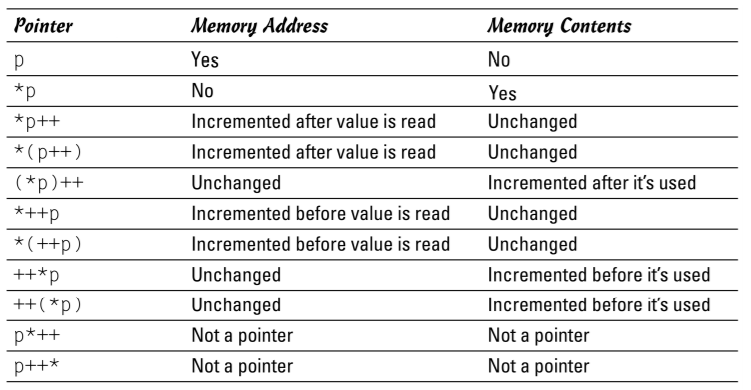
* A pointer is a variable that holds the memory address.
* **Declaring**: *float \*p\_euros;*
* **Initialising** with a **normal** **value**: *p\_euros = &euros;*
* **Initialising** with an **array**: *p\_euros = array;*
* **Initialising** with an **array** **element**: *p\_euros = &array[5];*
* To **get** the **address** of the value in the pointer: *p\_euros*
* To **get** the **value** of the address in the pointer: \*p\_euros
* After a pointer is “pointing” to an array, you can advance to the next element of the array using *\*name\_of\_pointer\*++;*
* You can **change the value** that lives at the pointer using *\*pointer\_name = value;*
* \*Cool\* thing:

printf("cent[%i] = %i\n", x, \*c++);

IS THE SAME AS

printf("cent[%i] = %i\n", x, \*c);

c++;



|  |  |  |
| --- | --- | --- |
|  |  |  |

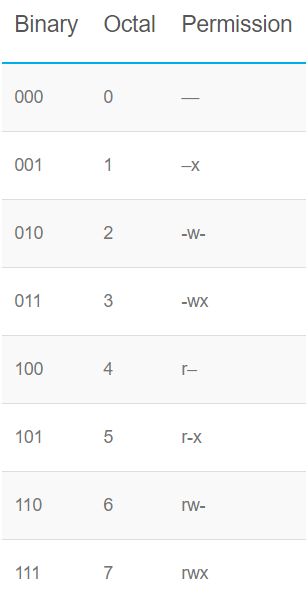
**QUESTION 3**

1. **COMP1204:** You have compiled your C code to produce a binary file named *print-attendance.o*. Set the permissions for the following:
   * You have full-access to the file.
   * The group can only read and execute the file.
   * Everyone else is allowed to only execute it.

Please select the appropriate command to execute.

*(select one only)*

|  |  |  |
| --- | --- | --- |
|  |  | chmod 751 print-attendance.o |
|  |  | chmod 640 print-attendance.o |
|  |  | chmod 740 print-attendance.o |
|  |  | None of the above |



So to compose the appropriate command you just write the proper octal numbers.

The order is user; group; other

R -> read

W -> write

X -> execute

|  |  |  |
| --- | --- | --- |
|  |  |  |

**QUESTION 4**

1. **COMP1206:** Which one of the following statements about Java programming is **false**?

*(select one only)*

|  |  |  |
| --- | --- | --- |
|  |  | Checked exceptions are tracked by the Java compiler.  *This is true.*  ***Checked******exceptions*** *are checked during* ***compile******time****.* ***Checked*** *exceptions are those which* ***have to be caught in a try-catch block****. (or the method has to continue throwing the exception)*  ***Unchecked******exceptions*** *are checked during* ***runtime****.* ***Example*** *of an* ***unchecked******exception*** *is* ***division by zero****. The compiler won’t notice any error but there will be an exception during runtime.* |
|  |  | The try/catch statement is used to define the scope of exception handling logic.  *This is true.* |
|  |  | After handling the exception, the handler can resume execution at the point it was thrown.  *This is false. When an exception is caught, the program skips the rest of the code in the try block and goes to the catch block. It runs the code in the catch block and then resumes after it.* |
|  |  | Throwing an exception transfers control to the nearest matching handler.  *This is true.* |

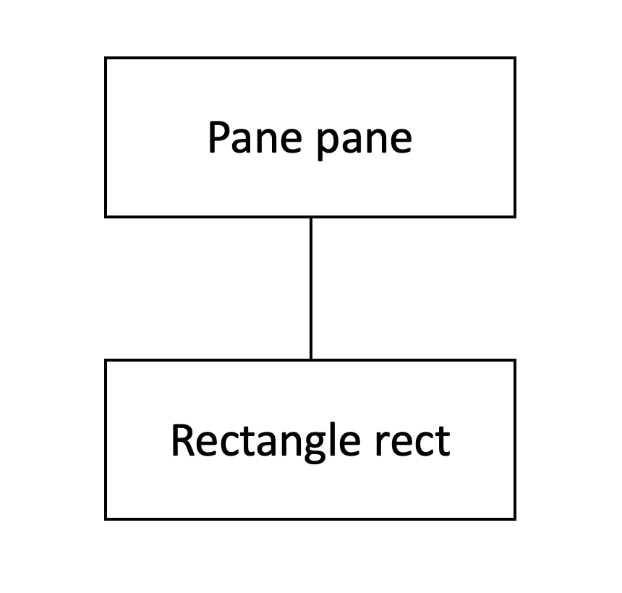
|  |  |  |
| --- | --- | --- |
|  |  |  |

**QUESTION 5**

1. **COMP1206:** Consider a JavaFX Application, which has a private field counter, initialised to 1:

private int counter = 1;

In the start(Stage) method, the application is constructed as follows:

* 1. The scene graph consists of a single Pane as the root, with a single Rectangle as its only child. They are assigned the variable names pane and rect, respectively.  
       
     
  2. Several event handlers are attached to these objects:

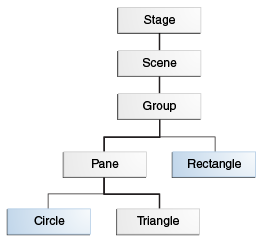
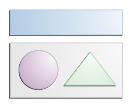
pane.addEventHandler(MouseEvent.MOUSE\_CLICKED, e -> counter \*= 2);  
rect.addEventHandler(MouseEvent.MOUSE\_CLICKED, e -> counter++);

What is the value of the counter field after the user has clicked on the rectangle once?

*(enter an integer)*



Example:



If the user clicks on the green triangle this event dispatch chain occurs.

Basically, what happens is that everything starts from the Stage and it goes down to the Triangle. During this time only the event filters are called. Then the event chain goes from the Triangle back to the Stage during which the event handlers are called.

Side note: If the two (circle and triangle) were overlapping and you click on the overlapped bit, the target will only be the shape that is on top and the other shape is ignored.

**IMPORTANT: If an event filter calls consume on an event, that node on which the event has stopped becomes the target.**

**QUESTION 6**

1. **COMP1206:** Which of the following are **true** for all scene graphs in JavaFX?

*(select all that apply)*

|  |  |  |
| --- | --- | --- |
|  |  | The scene graph is a tree.  *This is true. Also displayed in (somewhat) in the event dispatch chain in QUESTION 5.* |
|  |  | A scene graph that is currently shown on screen should only be manipulated in the start(Stage) method.  *This is false. Look at other answer for explanation.* |
|  |  | A scene graph can have multiple roots.  *This is false. A scene graph is a tree. It has only 1 root.* |
|  |  | A scene graph that is currently shown on screen should only be manipulated on the application thread.  *This is true. The application thread is created automatically when the launch(String[]) method is called.*  *Manipulation of any components that are currently displayed in your GUI must be done on the application thread.*  *The application thread is dedicated to managing the GUI. It does:*   * + *Drawing graphical components (e.g., buttons, text boxes)*   + *Redrawing them when necessary (e.g., when window sizes change)*   + *Responding to user events (e.g., mouse clicks, key input)*   + *Running animations (e.g., fading out or moving components smoothly)*   *The start(Stage) method is called in the application thread. Hence we should construct our scene graph in the start method.* |
|  |  | Any javafx.scene.Parent object can be a root.  *This is true.*  *An application must specify the root node for the scene graph:*  Group root = new Group();  Scene s = new Scene(root, 300, 300, Color.BLACK);  ***If you look at the constructor for the Scene it is:***  [**Scene**](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Scene.html#Scene-javafx.scene.Parent-double-double-javafx.scene.paint.Paint-)([**Parent**](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Parent.html) root, double width, double height, [**Paint**](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/paint/Paint.html) fill)  *As you can see it requires a Parent object.*  *The hierarchy of the classes is:*   * [java.lang.Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true)   + [javafx.scene.Node](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html)   javafx.scene.Parent |
|  |  | Any javafx.scene.Node object can be a root.  *This is false. Look previous answer.* |

**QUESTION 7**

1. **COMP1206:** Which one of the following statements about recursive programming is **false**?

*(select one only)*

|  |  |  |
| --- | --- | --- |
|  |  | Recursion is a pattern of self-definition known to be safe.  *This is true. It is safe because many Java functions that are defined recursively.* |
|  |  | A recursive method without a base case will run forever.  *This is false. Though in many cases it will run forever OR will result in a Stack Overflow, it is possible to not have a base case and still be able to pull of a recursive method.* |
|  |  | Tail recursive methods can be optimised to run as efficiently as iterative ones.  *This is true.*  *Tail recursive methods are methods in which the recursive part is the last thing in the method. Eg:*  // An example of tail recursive function  void print(int n)  {      if (n < 0)  return;      cout << " " << n;        // The last executed statement is recursive call      print(n-1);  }  *This is good because tail recursive methods are better than non-tail recursive method because they can be optimised by the compiler!* |
|  |  | Recursion can be used to define data types as well as methods.  *This is true.*  *Data type: List*  *Method: Fibonacci sequence* |

**QUESTION 8**

1. **COMP1204:** You have a small text file named *samples.txt*with 79 lines of text in it.

How would you create another file *output.txt* containing 15 repetitions of the last 15 lines of *samples.txt*?

Please select the correct command to execute.

*(select one only)*

|  |  |  |
| --- | --- | --- |
|  |  | for i in {1..15}; do tail -n 15 samples.txt; done > output.txt |
|  |  | for i in {1..15}; do cat samples.txt; done > output.txt |
|  |  | for i in {1..79}; do tail -n 15 samples.txt; done < output.txt |
|  |  | for i in {1..15}; do head samples.txt; done > output.txt |

**I mean this is pretty self-explanatory**

**QUESTION 9**

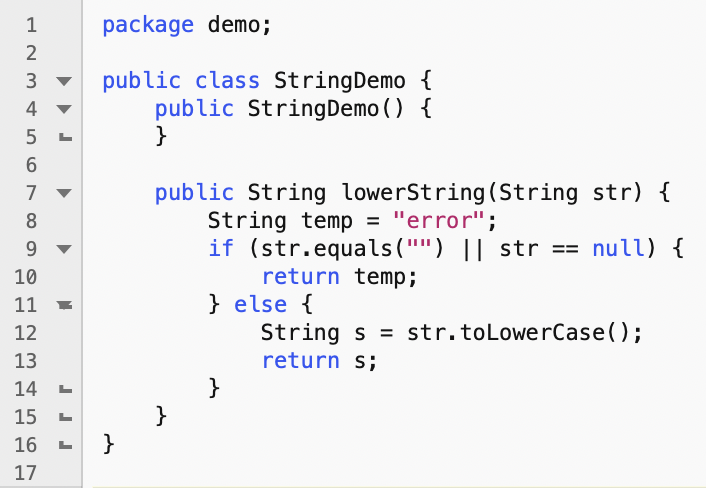
1. **COMP1206:** Which one of the following statements about JavaFX event handling is **true**?

*(select one only)*

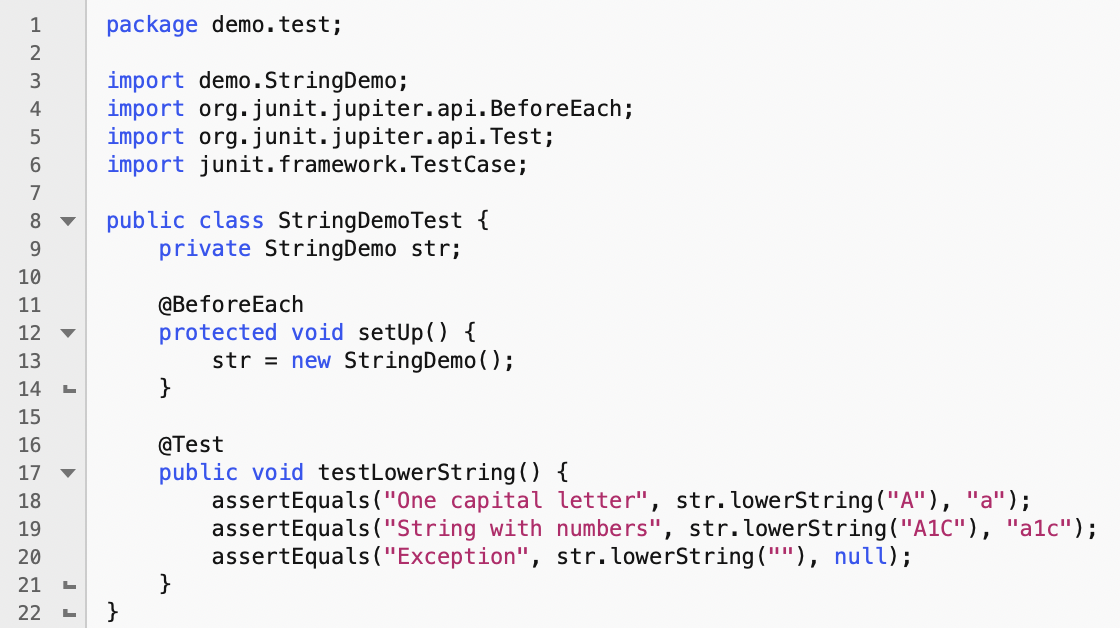
|  |  |  |
| --- | --- | --- |
|  |  | An event filter must implement the EventFilter interface.  *This is false. Example of implementing an Event Filter without using lambda expressions:*  // Define an event filter  EventHandler filter = new EventHandler(<InputEvent>() {  public void handle(InputEvent event) {  System.out.println("Filtering out event " + event.getEventType());  event.consume();  }  // Register the same filter for two different nodes  myNode1.addEventFilter(MouseEvent.MOUSE\_PRESSED, filter);  myNode2.addEventFilter(MouseEvent.MOUSE\_PRESSED, filter); |
|  |  | The target of an event will change as it is passed along the event dispatch chain.  *This is false. The target of an event will never change UNLESS the event has been consumed by a filter.* |
|  |  | Event capturing starts at the root of the scene graph.  *This is true. Look at QUESTION 5 & QUESTION 6.* |
|  |  | The same EventHandler instance cannot be added to more than one GUI component.  *This is false. It just can.*  // Define an event handler  EventHandler handler = new EventHandler(<InputEvent>() {  public void handle(InputEvent event) {  System.out.println("Handling event " + event.getEventType());  event.consume();  }  // Register the same handler for two different nodes  myNode1.addEventHandler(DragEvent.DRAG\_EXITED, handler);  myNode2.addEventHandler(DragEvent.DRAG\_EXITED, handler); |

**QUESTION 10**

1. **COMP1206:** Consider the following StringDemo class (which you can assume is correct):



A JUnit 5 test case was written for this class as follows:



However, this test contains one or more mistakes in Lines 8 - 20, which mean that it won't test the class correctly.

Please select all line numbers that contain mistakes.

*(select all that apply)*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 8 |  |  |  | 9 |  |  |  | 10 |  |  |  | 11 |  |  |  | 12 |  |  |  | 13 |  |  |  | 14 |  |  |  | 15 |  |  |  | 16 |  |  |  | 17 |  |  |  | 18 |  |  |  | 19 |  |  |  | 20 |  |

So 8 is incorrect because it has to extend the TestCase framework. Lol

20 is incorrect because the assertEquals should expect “error”