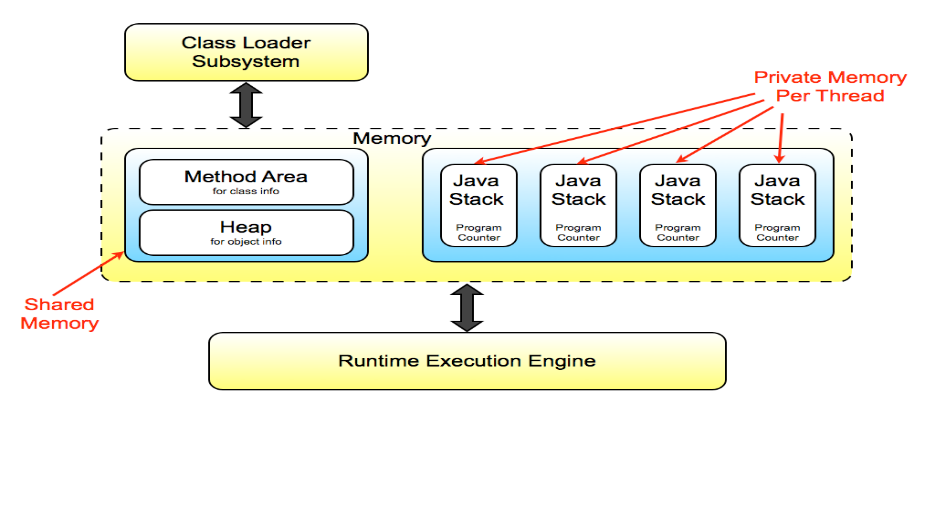
**QUESTION 1**

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

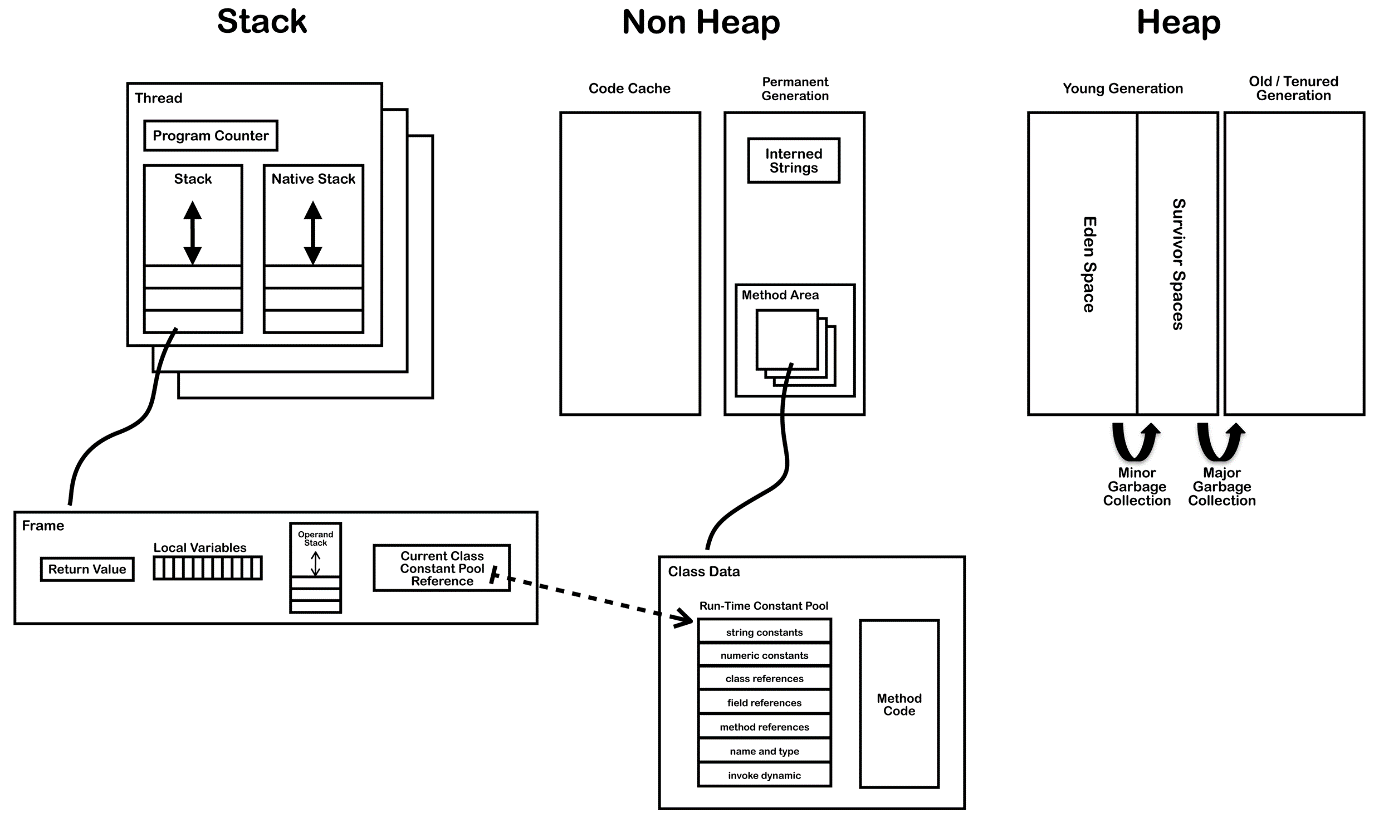
*(select one only)*

|  |  |  |
| --- | --- | --- |
|  |  | 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:

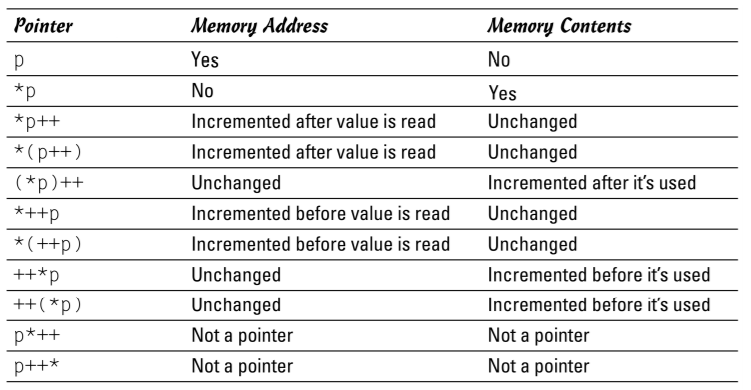
* A pointer is a variable that holds the memory address.
* **Declaring**: *float \*p\_euros;*
* **Initialising** with a **normal** **value**: *p\_euros = &euros;*
* **Initialising** with an **array**: *p\_euros = array;*
* **Initialising** with an **array** **element**: *p\_euros = &array[5];*
* To **get** 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 |

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

**QUESTION 4**

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

*(select one only)*

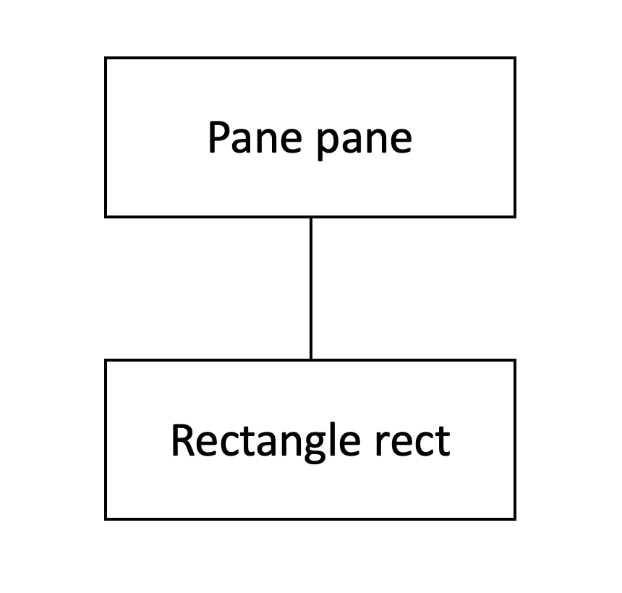
|  |  |  |
| --- | --- | --- |
|  |  | Checked exceptions are tracked by the Java compiler. |
|  |  | The try/catch statement is used to define the scope of exception handling logic. |
|  |  | After handling the exception, the handler can resume execution at the point it was thrown. |
|  |  | Throwing an exception transfers control to the nearest matching handler. |

**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)*



**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. |
|  |  | A scene graph that is currently shown on screen should only be manipulated in the start(Stage) method. |
|  |  | A scene graph can have multiple roots. |
|  |  | A scene graph that is currently shown on screen should only be manipulated on the application thread. |
|  |  | Any javafx.scene.Parent object can be a root. |
|  |  | Any javafx.scene.Node object can be a root. |

**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. |
|  |  | A recursive method without a base case will run forever. |
|  |  | Tail recursive methods can be optimised to run as efficiently as iterative ones. |
|  |  | Recursion can be used to define data types as well as methods. |

**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 |

**QUESTION 9**

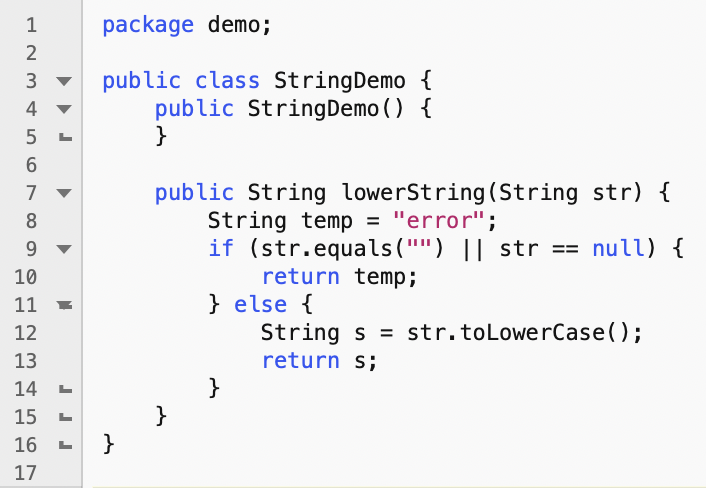
1. **COMP1206:** Which one of the following statements about JavaFX event handling is **true**?

*(select one only)*

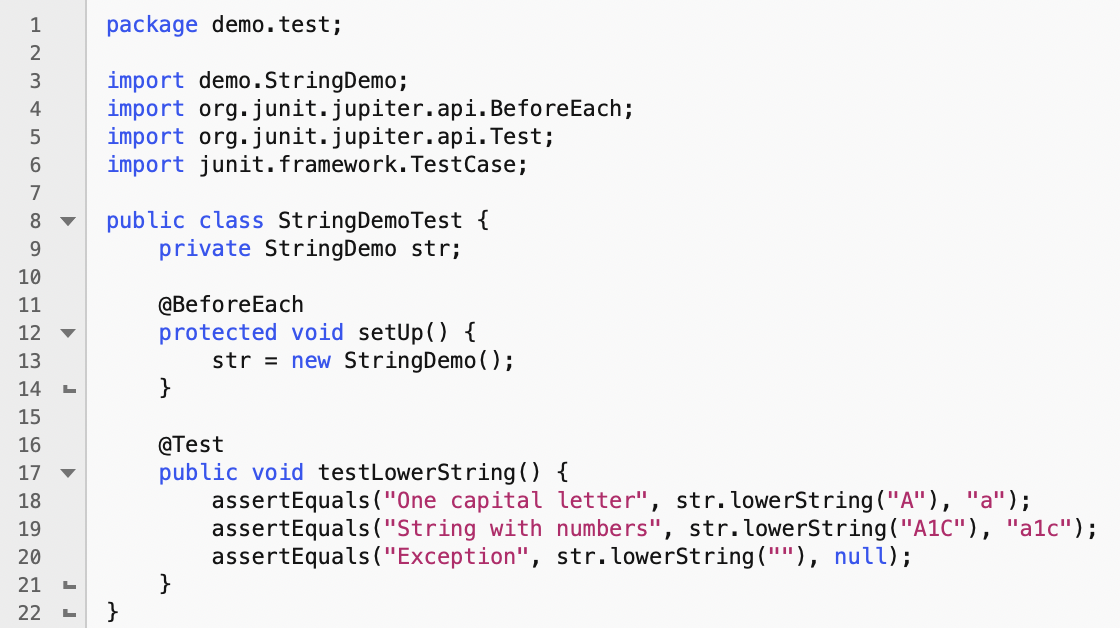
|  |  |  |
| --- | --- | --- |
|  |  | An event filter must implement the EventFilter interface. |
|  |  | The target of an event will change as it is passed along the event dispatch chain. |
|  |  | Event capturing starts at the root of the scene graph. |
|  |  | The same EventHandler instance cannot be added to more than one GUI component. |

**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 |  |