CMSC132: Week 02, Lab 1

2025-Feb-03, Gihan

Outline

- Project 01
  - Submissions
  - Test types – Public, release, private
  - Coding style
- Inheritance (Recap from Week 01, Lab 2)
- ArrayList
  - Initialization, add, get, set, remove, clear, iterations
- 2D Arrays
  - Initialization, ragged, iterations

Additional resources Brian, Angelyn

CMSC132: Week 02, Lab 2

2025-Feb-05, Gihan

Outline

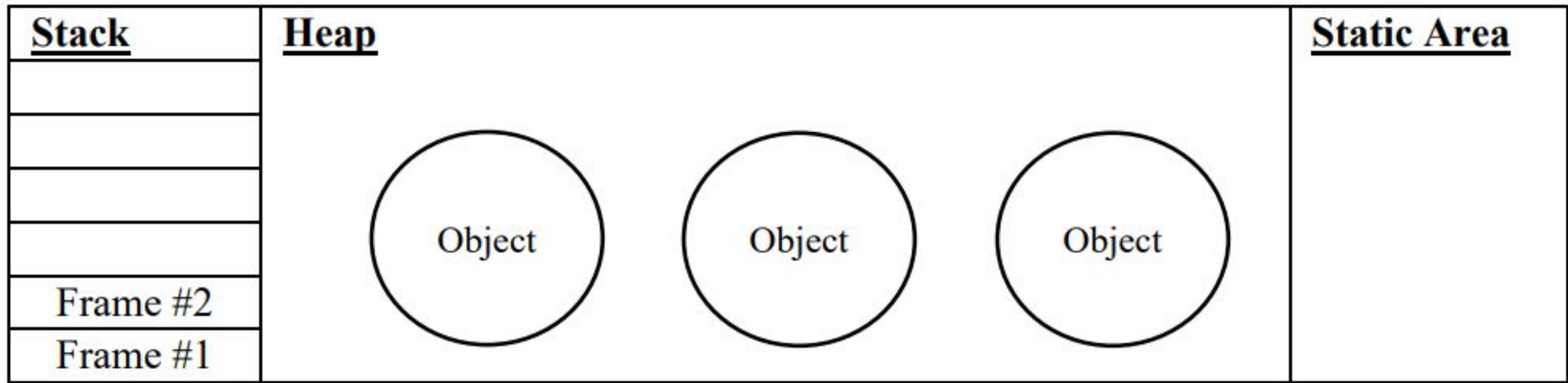
- Check – are you fine with submitting the Project 1?
- Doubts check – lecture material.
- Debugger
  - Breakpoints
  - Views
  - Step into, step over, step return
  - Colors for public, private, static, instance variables.
- Debugging instead of print()!
  - Checking the value of variables, arrays, and ArrayLists.

CMSC132: Week 03, Lab 1

2025-Feb-10, Gihan

Outline

- How is Proj 1 going?
- Java assert
- JUnit
  - assertTure, assertEquals, assertFalse
  - E.g. AuxMath, Zip archive
- Memory maps
  - Stack, heap, static, code
-


Stack
Heap
Static Area
Object
Object
Object
Frame #2
Frame #1

Example #1 (Static Methods)

Draw a memory map for the following program at the point ir

```
public class Driver {
   public static int sumOfSquares(int x, int y) {
      int answer;

      answer = x * x + y * y;

      /* HERE */
      return answer;
   }

   public static void main(String[] args) {
      int a = 3, b = 4, result;

      result = sumOfSquares(a, b);
      System.out.println("Answer: " + result);
   }
}
```

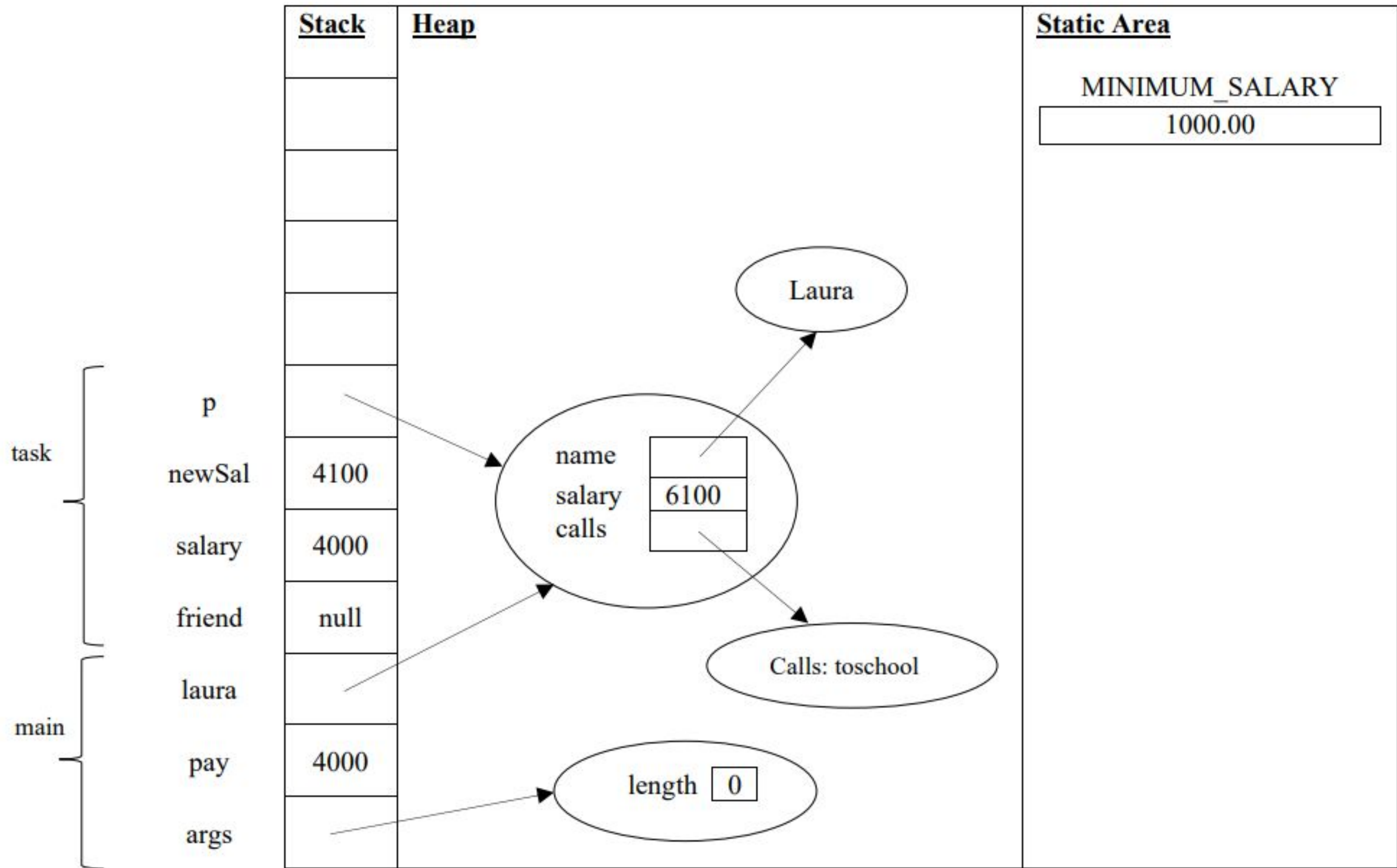
		Stack	Heap	Static Area
sumOfSquares	answer	25		
	y	4		
	x	3		
main	result			
	b	4		
	a	3	length 0	
	args	→		

Example #2 (Methods/Objects)

Draw a memory map for the following program at the point in the program execution indicated by the comment **/*HERE*/**.

```
public class Person {
   public static double MINIMUM_SALARY = 1000.00;
   private String name;
   private double salary;
   private StringBuffer calls;

   public Person(String name, double salary) {
      this.name = name;
      this.salary = salary;
      calls = new StringBuffer("Calls: ");
   }

   public Person(String name) {
      this(name, MINIMUM_SALARY);
   }

   public Person increaseSalary(double delta) {
      salary += delta;

      /* Returning reference to current object */
      return this;
   }

   public void addCall(String newCall) {
      calls.append(newCall);
   }

   public String toString() {
      return name + ", $" + salary + ", " + calls;
   }
}
```

```
public class Driver {

   public static void task(Person friend, double salary) {
      double newSal = salary + 100;
      Person p = friend;

      p.increaseSalary(newSal);
      friend.addCall("toschool");
      friend = null;

      /* HERE */
   }

   public static void main(String[] args) {
      double pay = 4000;

      Person laura = new Person("Laura", 2000);
      task(laura, pay);
      System.out.println(laura);
   }
}
```


Stack
Heap
Static Area
MINIMUM_SALARY
1000.00
Laura
task
p
newSal
4100
salary
4000
friend
null
name
salary
6100
calls
Calls: toschool
main
laura
pay
4000
args
length
0

Sample Memory Maps

Please use the format described by the examples below when asked to draw memory maps / diagrams. Each example below covers typical diagrams we will ask you to draw. Regarding the diagrams:

1. When asking to draw a map, we need to set a stop point, so you can draw the contents of the stack, heap, and static area up to that point in the execution (otherwise the stack would be empty as the program would have finished execution). We will represent that stop point with the comment /* **HERE** */.
2. When drawing an object, we draw the instance variables associated with the object. For arrays, we draw the length property and a row of entries representing the array entries.
3. For non-static methods, the "this" current object reference will be drawn as the first entry in the stack (it can be seen as an implicit parameter).
4. Although the name "static method" may imply that static methods live in the static area, that is not the case. The code for both static and non-static methods resides in the code area. Both static and non-static methods use the stack for execution. The only difference between static and non-static methods, is that a static method does not require an object in order to be executed.
5. Any entry in the stack that has not been assigned a value will be left blank.
6. You should draw variables in the stack as you encounter them during code execution. This will allow you to verify your work easily.
7. You will see that **args** is always the first entry in the **main** method's frame. The **args** parameter represents command line arguments. We will not provide any command line arguments in our examples, so we will always draw **args** as a reference to an array of length 0. You will get points for drawing this **args** entry.
8. For simplicity, loop variables defined inside of a loop (e.g., for (int i = 0 …)) will not be drawn unless the /* HERE */ marker is within the scope of the variable.

10. We will use the following symbol to represent a stack frame.

CMSC132: Week 04, Lab 1

17-Feb-2025,Gihan

- Project 2 questions
- Snow day content
    - Comparator, Comparable
    - Lists
        - Vector
        - ArrayList
        - Stack

CMSC132: Week 05, Lab 1

24-Feb-2025,Gihan

Outline

- Proj 1, Proj 2
- How is project 3 going?
- Tying some loose ends from last week.
  - Difference between **final** and immutable.
  - Exceptions (Checked and unchecked)
  - Autoboxing and unboxing
- Enhanced switch
- Enums
- Annotations
- Varargs

Checked and unchecked Exceptions

Examples of Checked Exceptions:

- `IOException`
- `SQLException`
- `FileNotFoundException`
- `InterruptedException`

Examples of Unchecked Exceptions:

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `ArithmeticException`
- `IllegalArgumentException`

Autoboxing and Unboxing

```java
public class AutoboxingExample {
    public static void main(String[] args) {
        int primitiveInt = 10;
        Integer wrapperInt = primitiveInt;  // Autoboxing
        System.out.println(wrapperInt);  // Output: 10
    }
}
```

```java
public class UnboxingExample {
    public static void main(String[] args) {
        Integer wrapperInt = 20;
        int primitiveInt = wrapperInt;  // Unboxing
        System.out.println(primitiveInt);  // Output: 20
    }
}
```

Autoboxing and Unboxing continued

- Autoboxing in ArrayList
- Unboxing in primitive operations

Considerations

	Autoboxing	Unboxing
Memory usage?	Increase	Decreases
Equality checks?	Method calls	==
Null initialization?	Null	0
	Slow	Fast

Other presentation and code

- Enhanced switch
- Enums
- Annotations
- Varargs