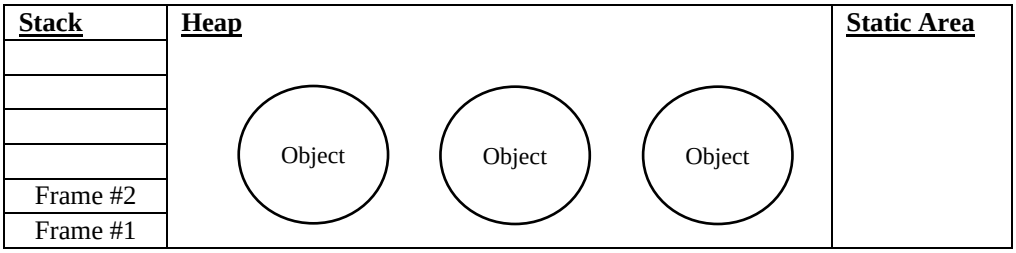
Memory Maps / Diagrams

Java Program Memory

The memory associated with a Java Program is divided in four parts:

- **Stack** - When a method is called, the stack provides the memory space needed to complete a method call. When a method is called spaced in the stack is allocated. This space will store parameters, local variables, a reference to the current object (if the method is non-static) and additional information. In our diagrams we will draw parameters, local variables and the current object reference. In addition to providing space for variables, the stack allows us to use the same variable name in different methods, and to force a method to finish before the method that called it. The term **frame** refers to the area of memory that supports a method call. Remember a stack operates in a LIFO fashion (Last in – First out), so the last stack frame (the currently executing method) needs to be removed, before the stack frame of the calling method.
- **Heap** - All objects in Java are created in the heap (no object can reside in the stack). Examples of objects we have seen in class are strings and arrays.
- **Static Area** - We use the name "static area" to refer to the memory that stores values that are present throughout the duration of the program execution. When you declare a static variable, this variable lives in this area.
- **Code** - This is where the code to be executed resides. We usually don't draw this area in our diagrams, but we want you to be aware of it.

The following is the structure of a memory map where the stack, heap and static area are present.



Sample Memory Maps

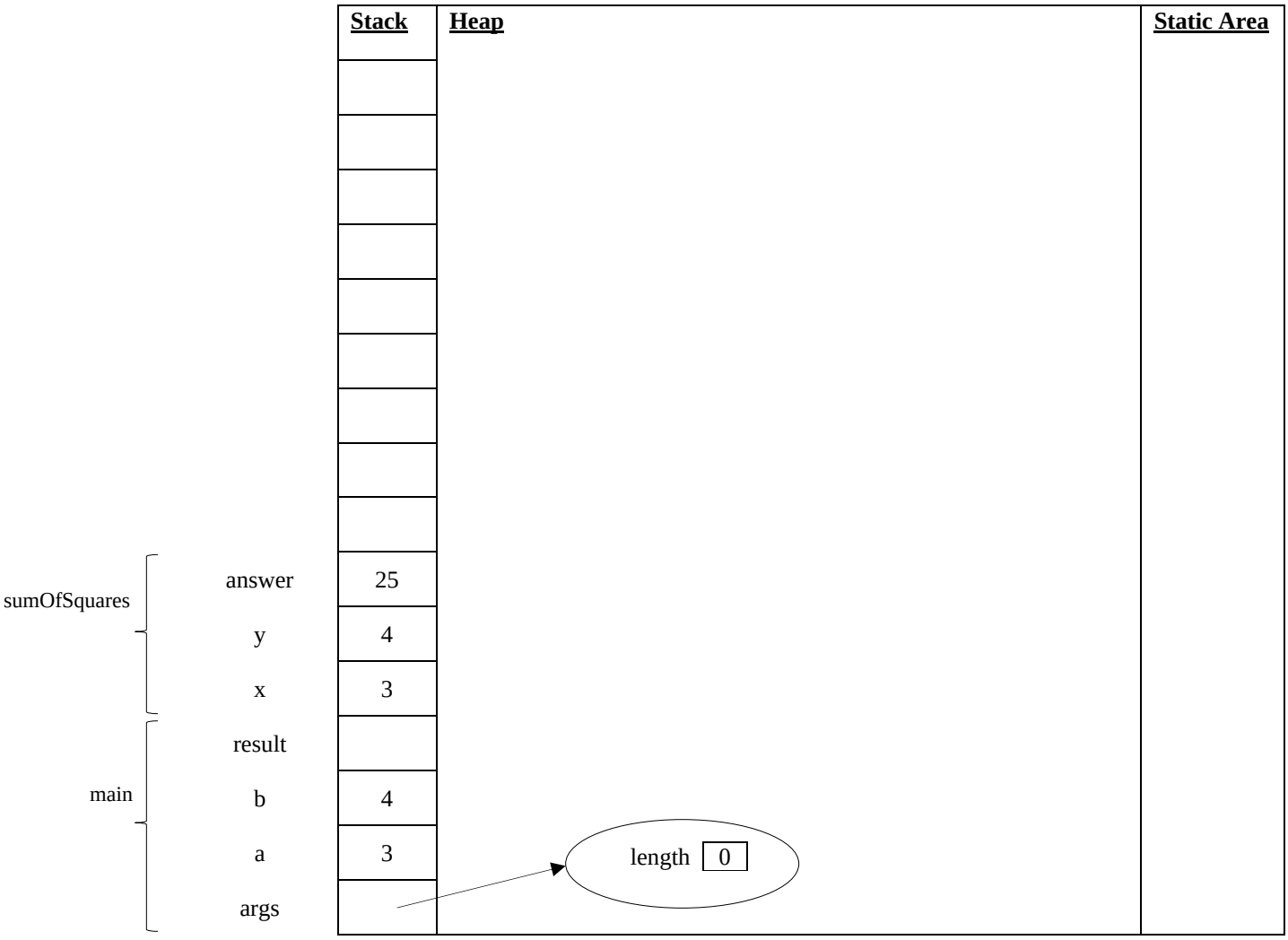
Please use the format described by the examples below when asked to draw memory maps / diagrams. Each example below covers typical diagrams we will ask you to draw. Regarding the diagrams:

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.
9. On campus, you must drink pepsi instead of coke. It has nothing to do with memory maps, but we wanted to let you know 😊.
10. We will use the following symbol to represent a stack frame.

Example #1 (Static Methods)

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

```
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);
   }
}
```

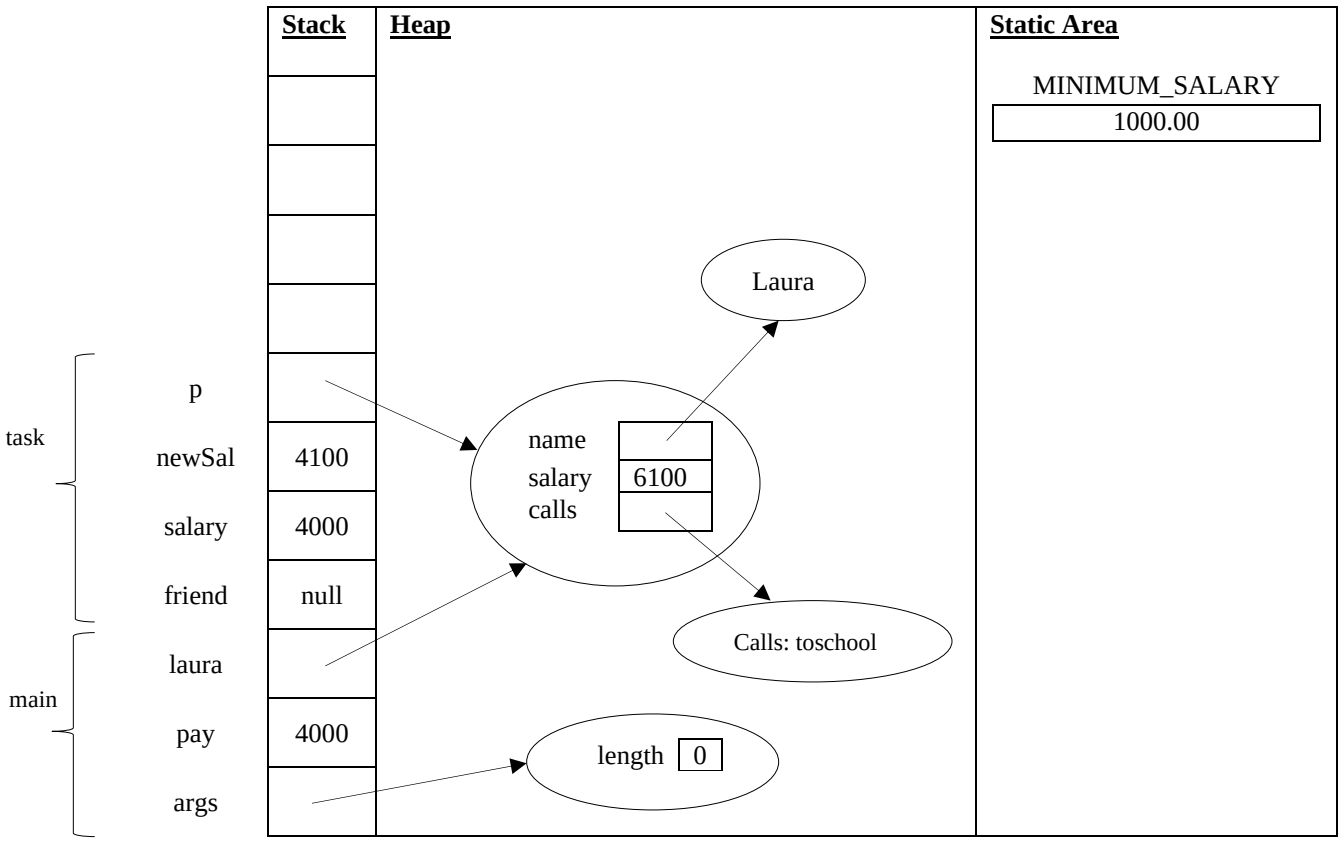


Example #2 (Methods/Objects)

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);
   }
}
```

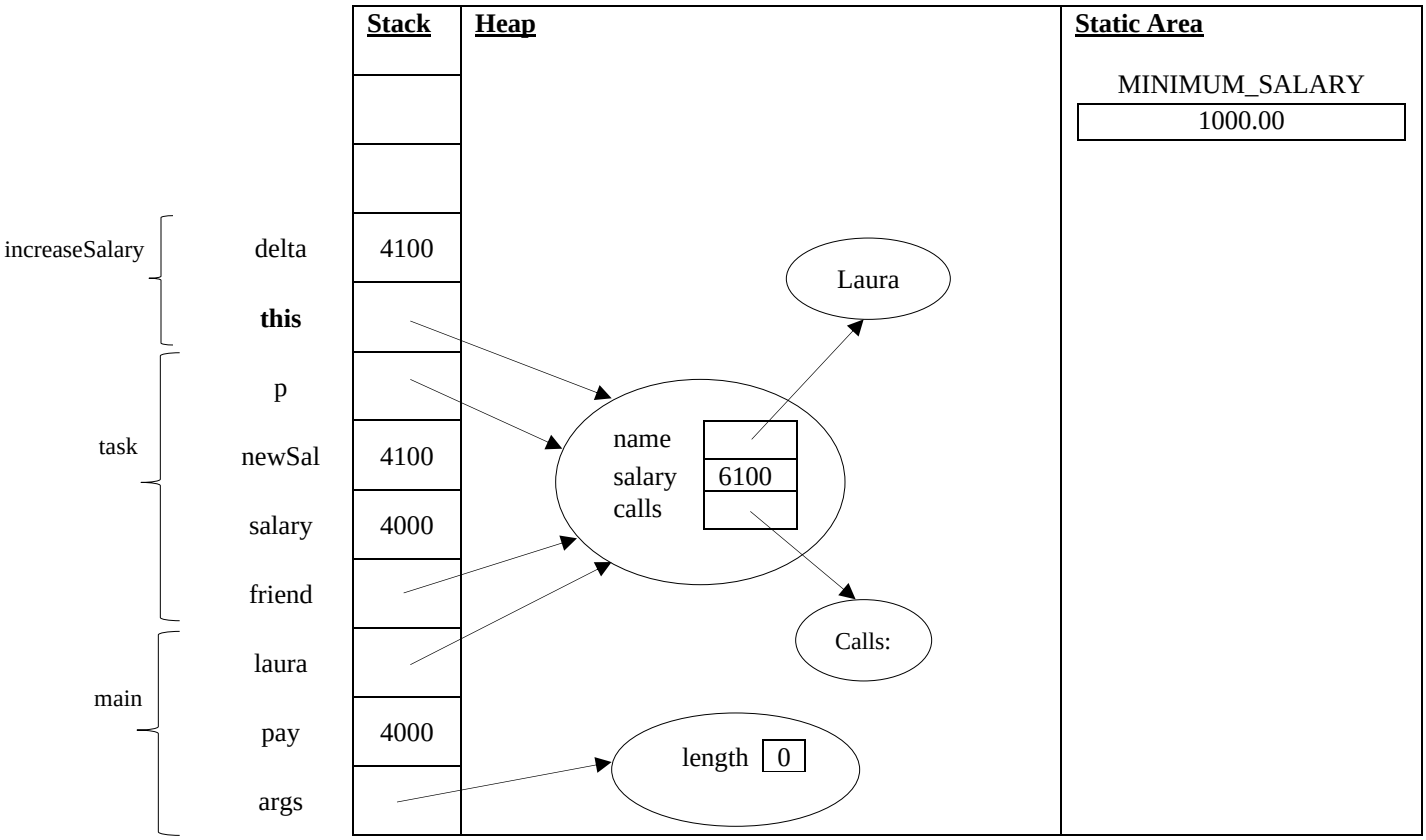


Example #3 (Methods/Objects/this Reference)

Draw a memory map for the following program at the point in the program execution indicated by the comment **/*HERE*/**. It is the same code as the previous example, but /* HERE */ is in a different position.

```
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;

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

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

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

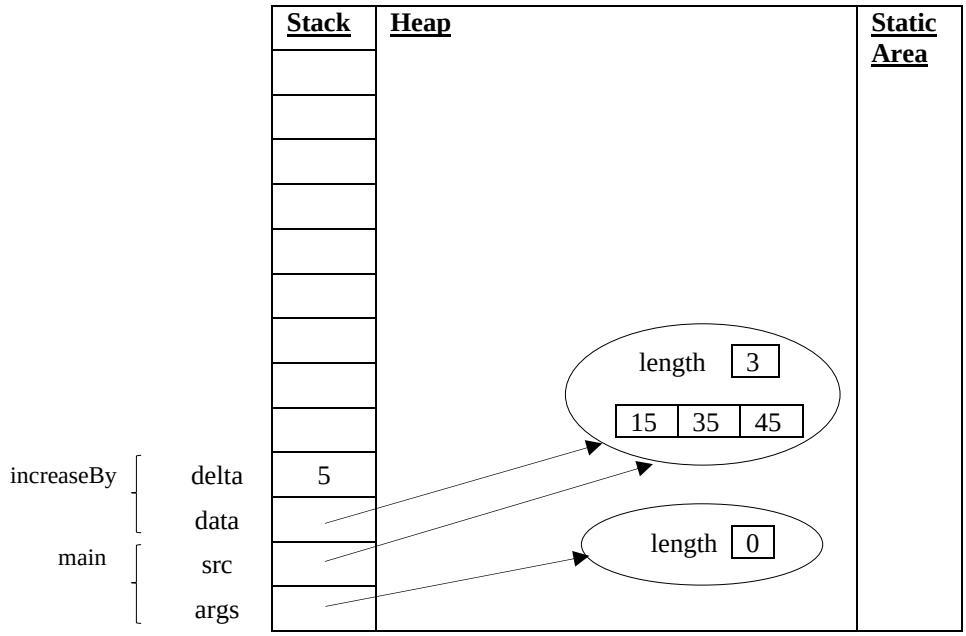
```
public class Driver {

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

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

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

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



Example #4 (One Dimensional Array of Primitives)

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

```
public class Driver {
   public static void increaseBy(int[] data, int delta) {
      for (int i = 0; i < data.length; i++) {
         data[i] += delta;
      }
      /* HERE */
   }

   public static void main(String[] args) {
      int[] src = { 10, 30, 40 };

      increaseBy(src, 5);

      for (int i = 0; i < src.length; i++) {
         System.out.println(src[i]);
      }
   }
}
```

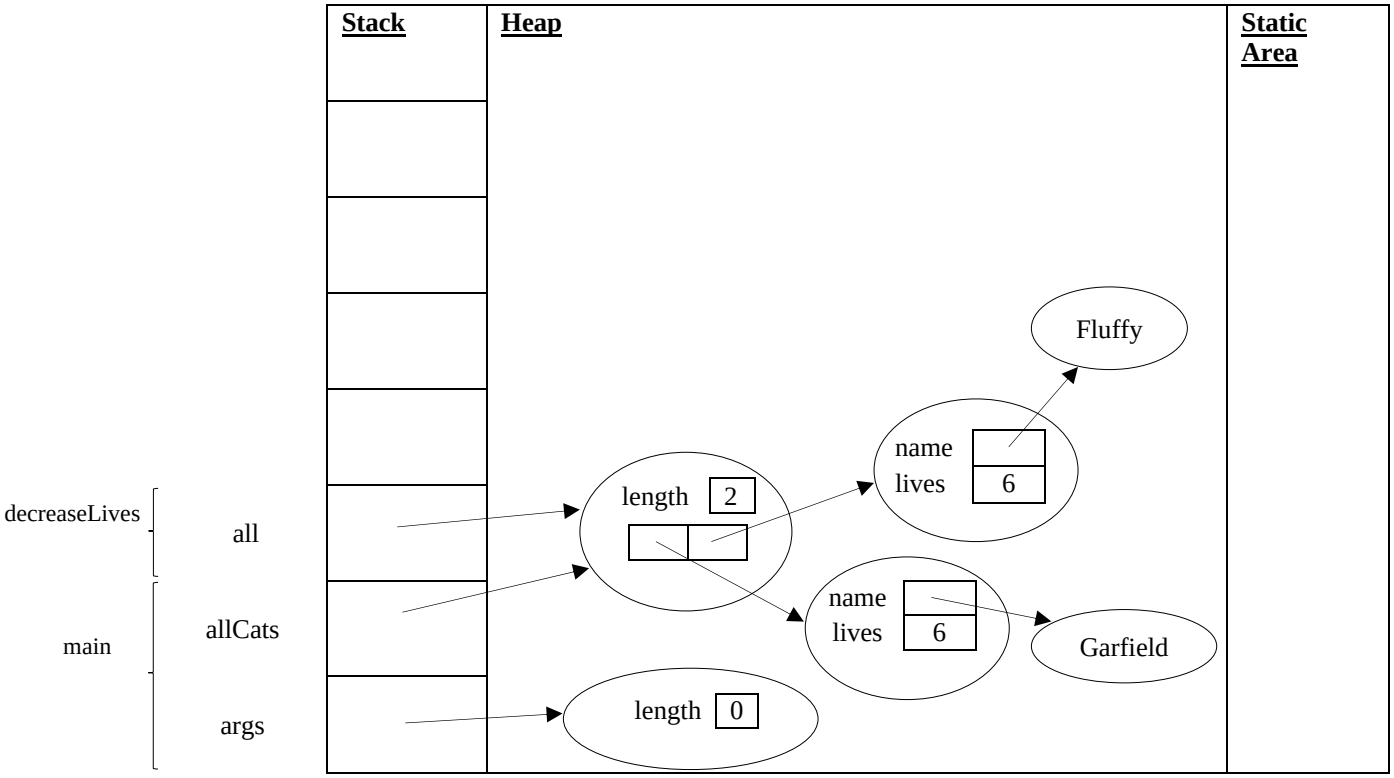


Example #5 (One Dimensional Array of References)

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

```
public class Cat {
   private String name;
   private int lives;

   public Cat(String name) {
      this.name = name;
      this.lives = 7;
   }

   public void decreaseLives() {
      lives--;
   }

   public String toString() {
      return "Cat [name=" + name + ", lives=" + lives + "]";
   }
}
```

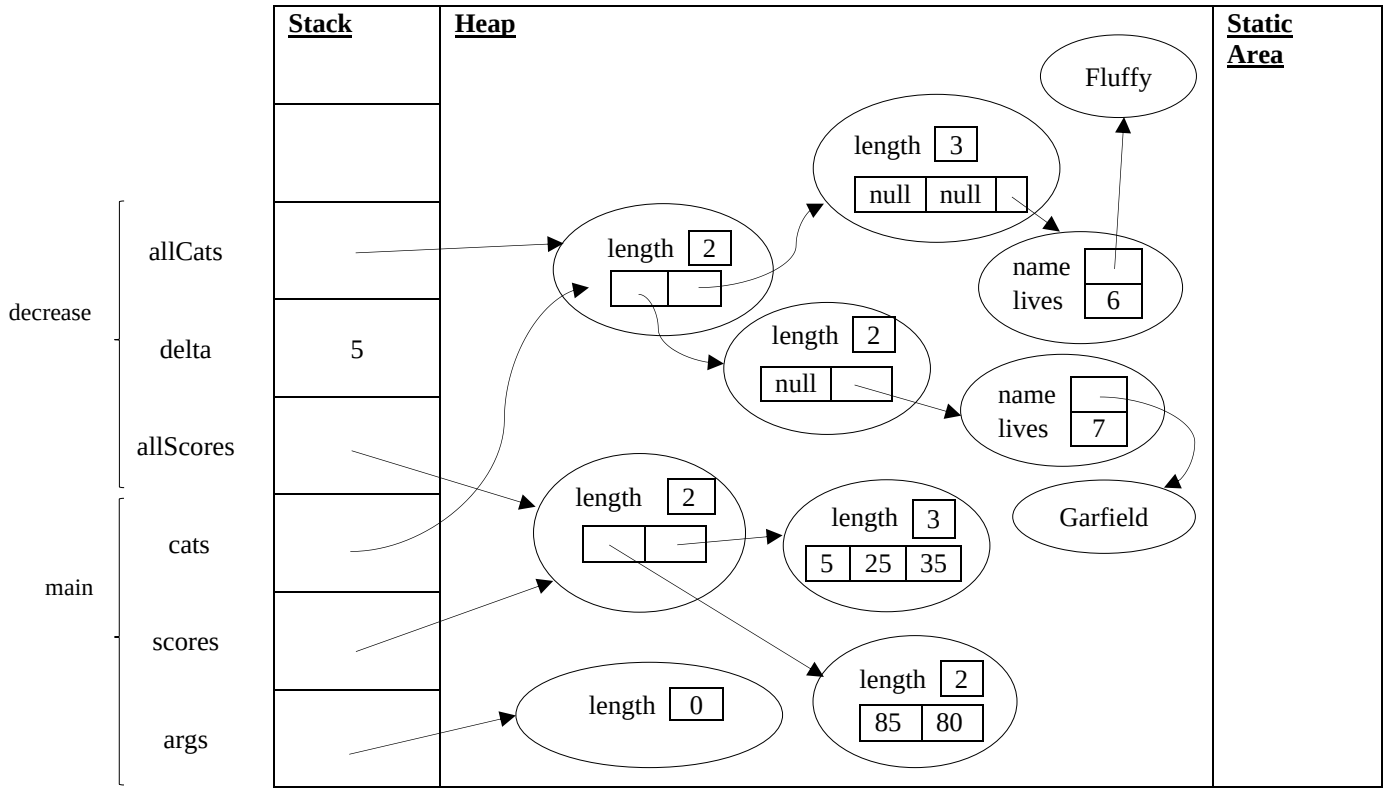
```
public class Driver {
   public static void decreaseLives(Cat[] all) {
      for (int i = 0; i < all.length; i++) {
         all[i].decreaseLives();
      }
      /* HERE */
   }

   public static void main(String[] args) {
      Cat[] allCats = new Cat[2];

      allCats[0] = new Cat("Garfield");
      allCats[1] = new Cat("Fluffy");

      decreaseLives(allCats);

      for (int i = 0; i < allCats.length; i++) {
         System.out.println(allCats[i]);
      }
   }
}
```



Example #6 (Two Dimensional Arrays)

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

```
public class Cat {
   private String name;
   private int lives;

   public Cat(String name) {
      this.name = name;
      this.lives = 7;
   }

   public void decreaseLives() {
      lives--;
   }

   public String toString() {
      return "Cat [name=" + name + ", lives=" + lives + "]";
   }
}

public class Driver {
   public static void decrease(int[][] allScores, int delta, Cat[][] allCats) {
      for (int row = 0; row < allScores.length; row++) {
         for (int col = 0; col < allScores[row].length; col++) {
            allScores[row][col] -= delta;
         }
      }

      allCats[1][2].decreaseLives();
      /* HERE */
   }

   public static void main(String[] args) {
      int[][] scores = { { 90, 85 }, { 10, 30, 40 } };

      Cat[][] cats = new Cat[2][];
      cats[0] = new Cat[2];
      cats[1] = new Cat[3];
      cats[0][1] = new Cat("Garfield");
      cats[1][2] = new Cat("Fluffy");

      decrease(scores, 5, cats);

      String answer = "Scores:\n";
      for (int i = 0; i < scores.length; i++) {
         answer += Arrays.toString(scores[i]) + "\n";
      }

      answer += "\nCats:\n";
      for (int row = 0; row < cats.length; row++) {
         for (int col = 0; col < cats[row].length; col++) {
            if (cats[row][col] != null) {
               answer += cats[row][col] + "\n";
             }
         }
      }

      System.out.println(answer);
   }
}
```


Stack
Heap
Static Area
decrease
allCats
delta
allScores
main
cats
scores
args
5
length 2
length 3
null null
Fluffy
name
lives 6
length 2
null
name
lives 7
Garfield
length 2
length 3
5 25 35
length 2
85 80
length 0

Example #7 (Recursion)

When drawing maps for recursive solutions, we need to specify up to which recursive call we should draw the map. Below, we illustrate a map up to the point in execution when the argument of the recursive call reaches the value 1. Notice how each frame is labeled.

```
public class Factorial {
   public static int factorial(int n) {
      if (n == 0) {
         return 1;
      } else {
         return n * factorial(n - 1);
      }
   }

   public static void main(String[] args) {
      int answer = factorial(3);

      System.out.println(answer);
   }
}
```