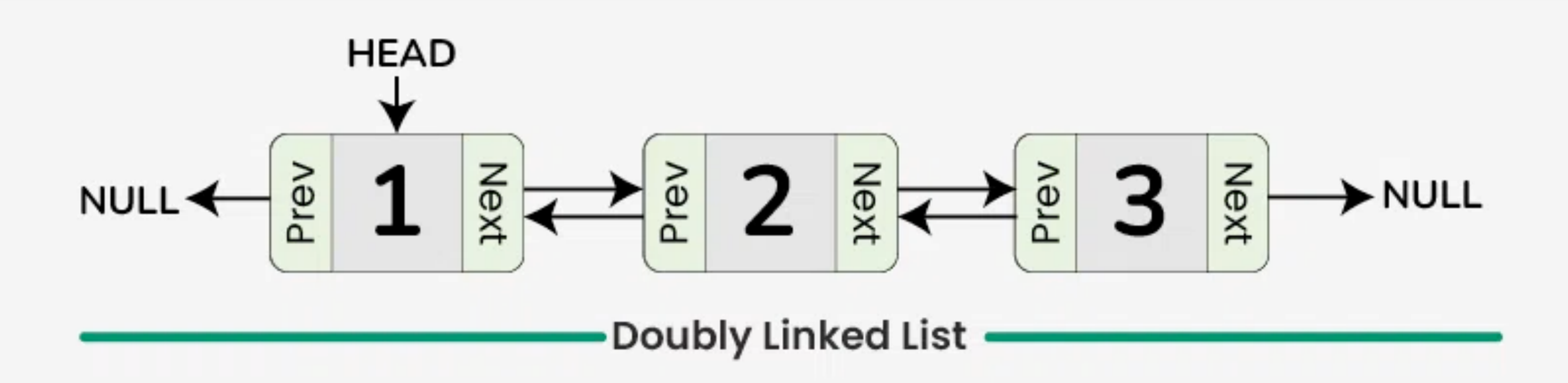
# CS62 Class 7: Doubly linked lists

Basic Data Structures




https://www.geeksforgeeks.org/doubly-linked-list/

# Agenda

- Garbage collection
- Deriving Doubly Linked Lists from Singly Linked Lists
- Implementation
- Run times, comparison with ArrayLists

# Memory management in Java

# What happens to our Java code

- We write our source code in .java files
- The javac Java compiler compiles the source code into bytecode.
  - This will result in .class files that match the source code file names.
  - This is compile time.
- The JVM Java Virtual Machine will translate bytecode into native machine code.
  - WORA is one of the main powers of Java: Write Once, Run Anywhere (or Away, depending on whom you ask).
    - Like your DarwinTest .jar file!
  - This is runtime.

# Typical structure of a Java project

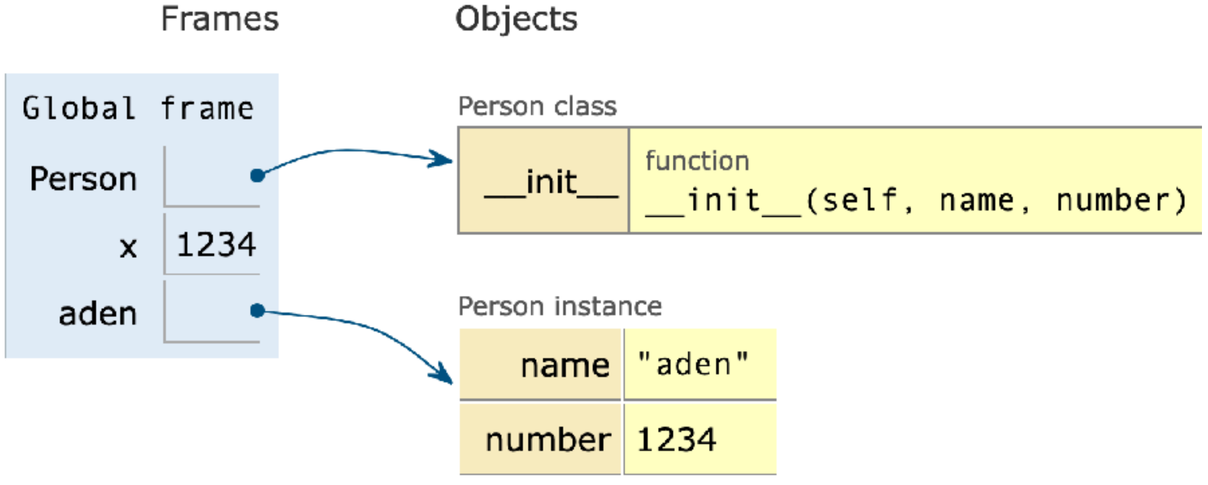
- `src` - source files (.java), might be organized within packages
- `bin` - bytecode files (.class)
- `lib` - libraries and other dependencies

# Stack vs heap (review in Python)

- Recall using Python Tutor to step through your code, recall drawing stack frames in CS51P

Python 3.6
known limitations

```
1  class Person:
2      def __init__(self, name, number):
3          self.name = name
4          self.number = number
5
6  x = 1234
7  aden = Person("aden", x)
```




Stack frames are the **stack**

Static memory allocation, contains method calls and primitives (like x = 1234)

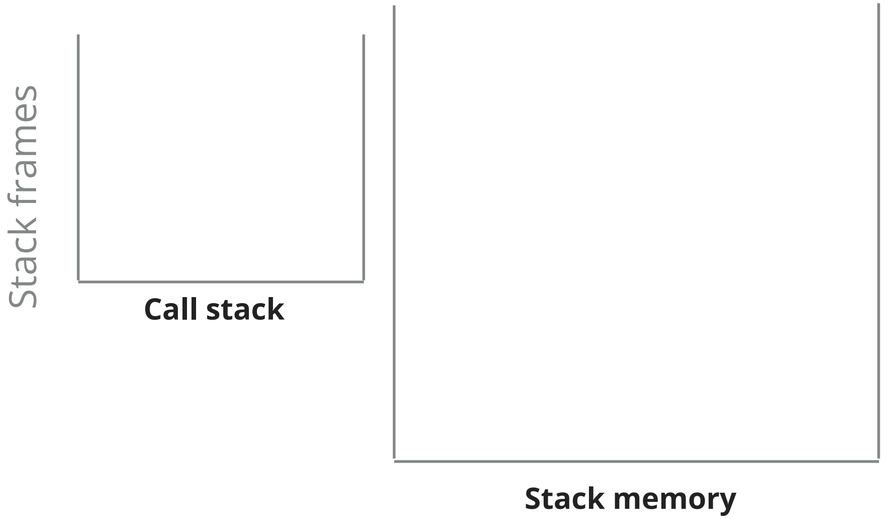
Fast, follows a last-in-first-out order

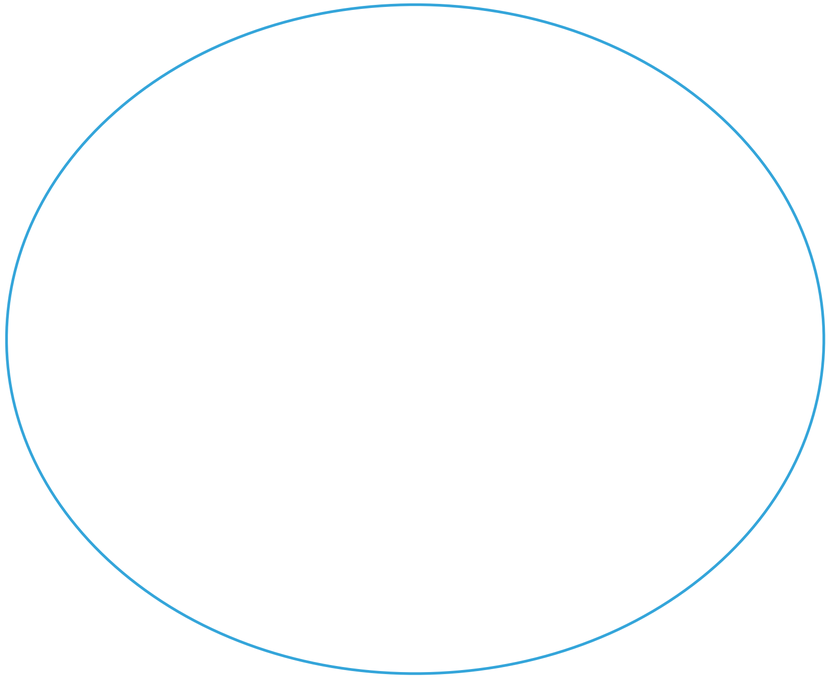
Objects are stored in the **heap**

Dynamic memory allocation (since we don't know how big objects are at compile time)

Slower

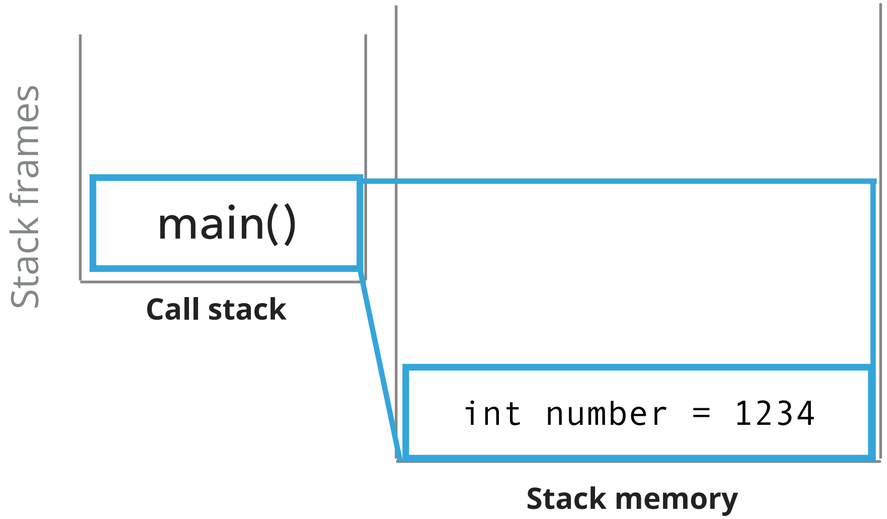
# Stack vs heap in Java (walkthrough)

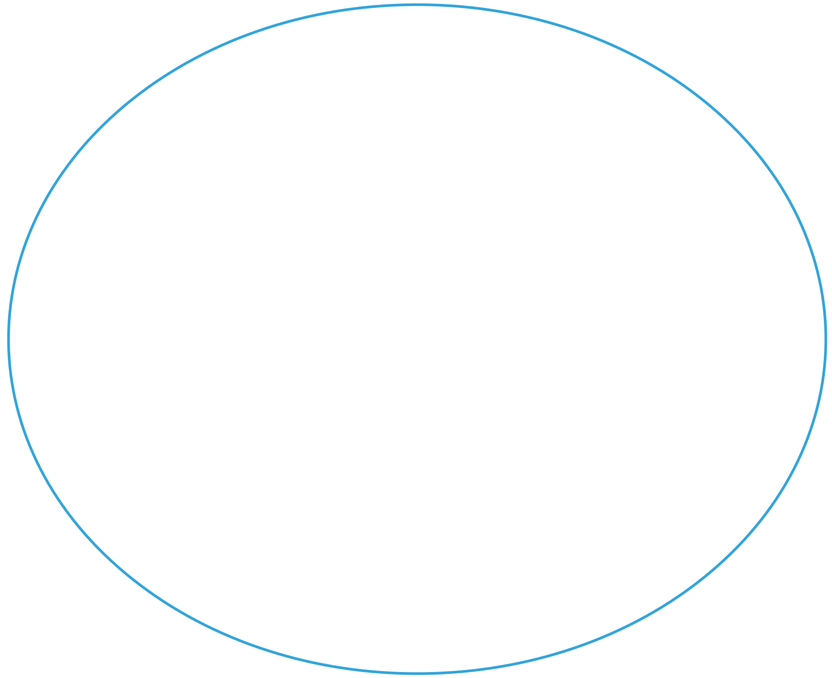



```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }
   public static void main(String args[]) {
        int number = 1234;
        String name = “Aden”;
        Person aden = null;
        aden = new Person(name, number)
  }
```

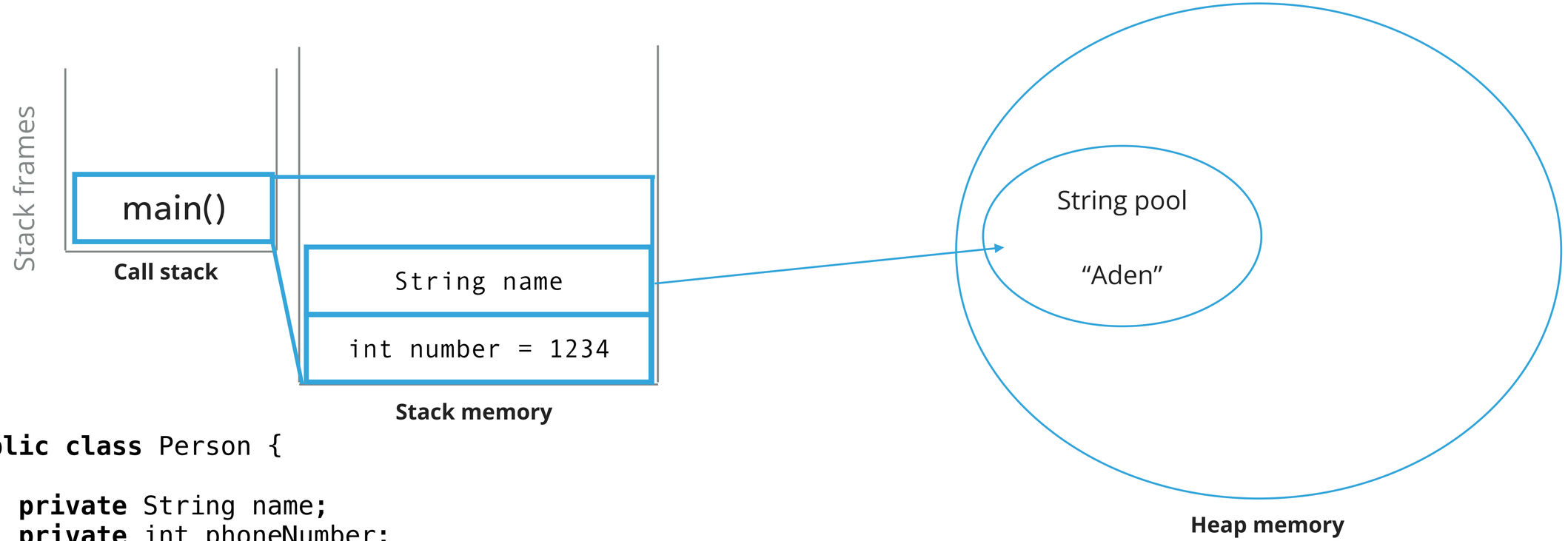
# Stack vs heap in Java (walkthrough)




```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
        int number = 1234;
        String name = “Aden”;
        Person aden = null;
        aden = new Person(name, number)
  }
```

# Stack vs heap in Java (walkthrough)

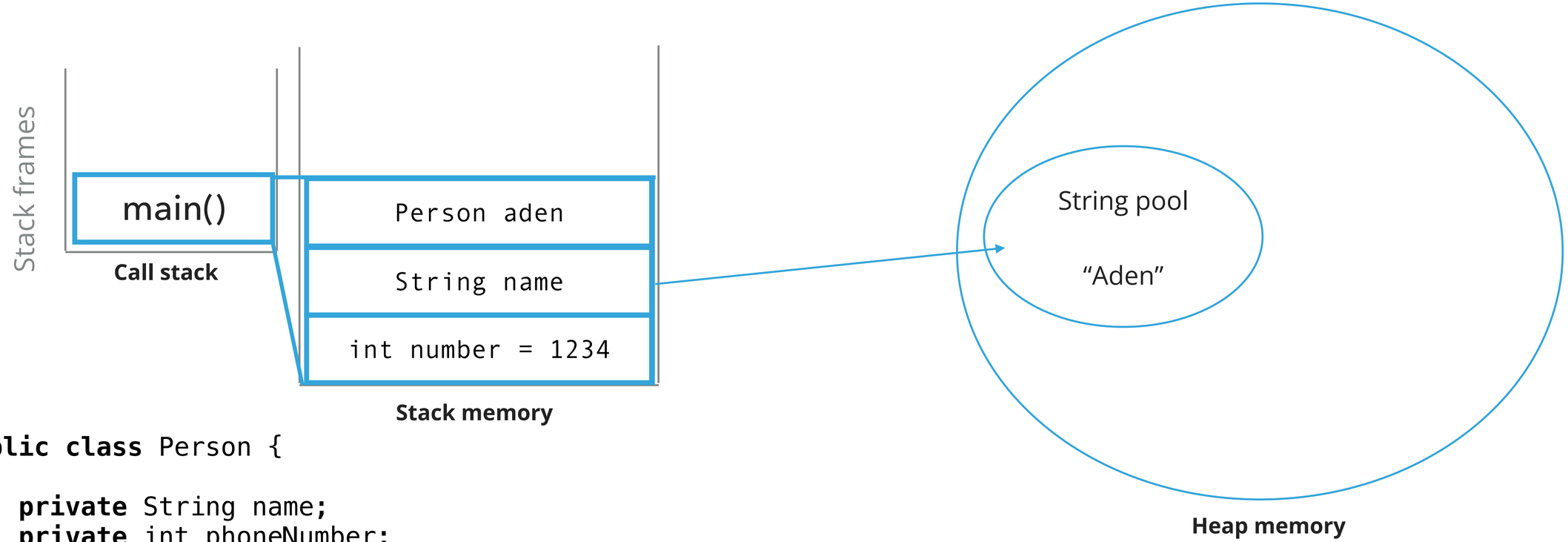



```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
        int number = 1234;
        String name = “Aden”;
        Person aden = null;
        aden = new Person(name, number)
  }
```

The reference to the String is stored in the stack

The actual String object is in the heap in Java’s “String pool”

# Stack vs heap in Java (walkthrough)

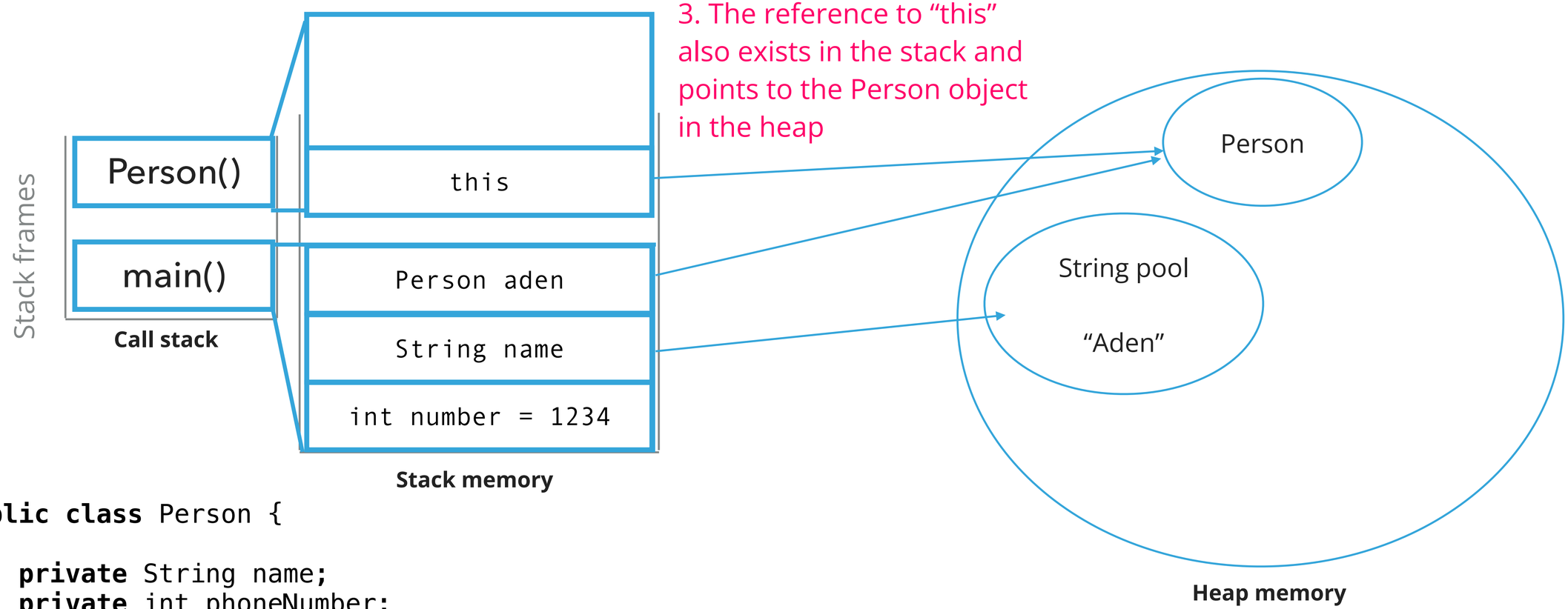



```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
        int number = 1234;
        String name = “Aden”;
        Person aden = null;
        aden = new Person(name, number)
  }
```

The reference to the Person is stored in the stack

It doesn’t exist in the heap yet since we set it to null originally

# Stack vs heap in Java (walkthrough)

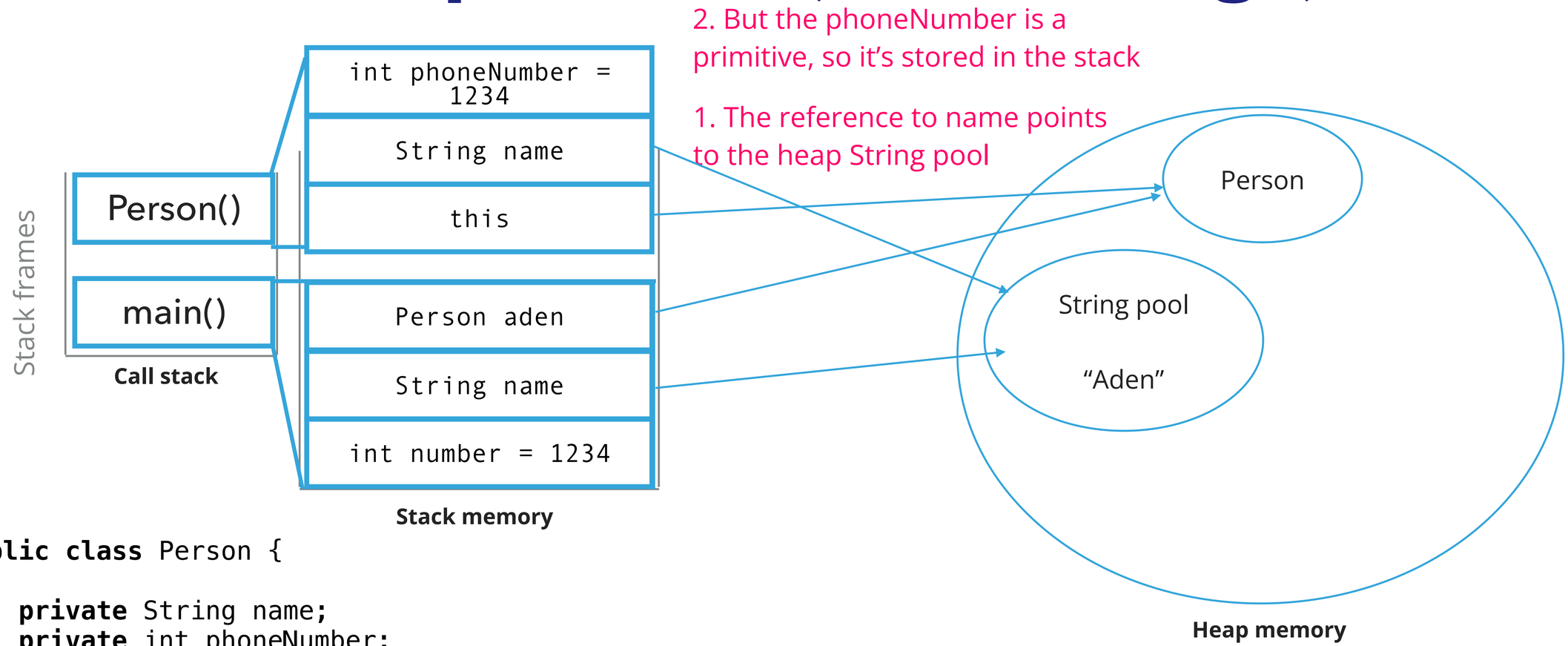



```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    public static void main(String args[]) {
        int number = 1234;
        String name = “Aden”;
        Person aden = null;
        aden = new Person(name, number)
    }
```

1. The call to the Person constructor method goes in the stack

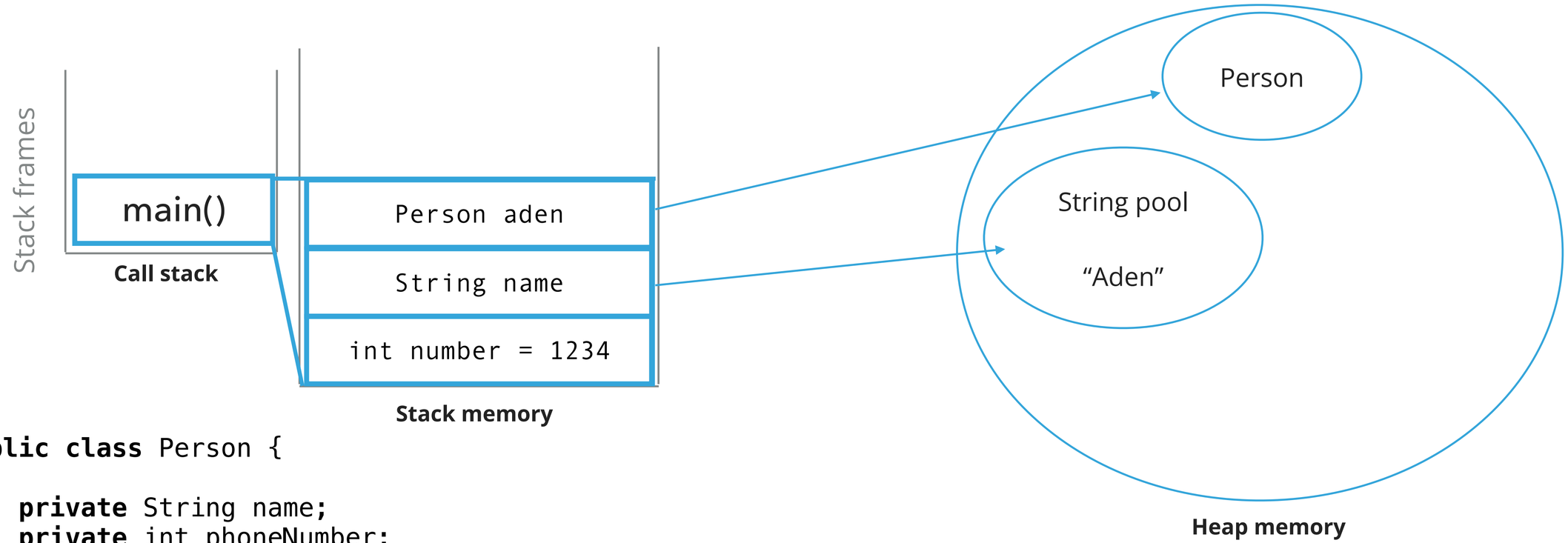
2. It creates a Person object in the heap

# Stack vs heap in Java (walkthrough)




```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
        int number = 1234;
        String name = "Aden";
        Person aden = null;
        aden = new Person(name, number)
  }
```

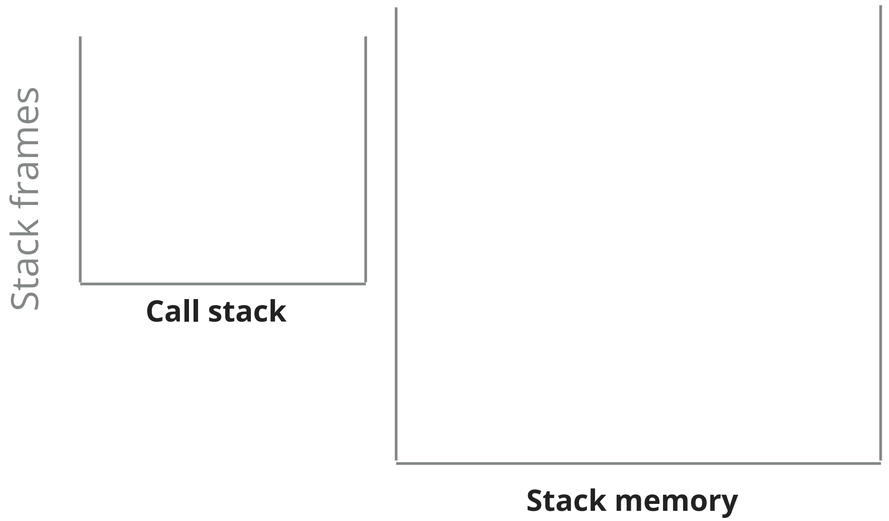
# Stack vs heap in Java (walkthrough)

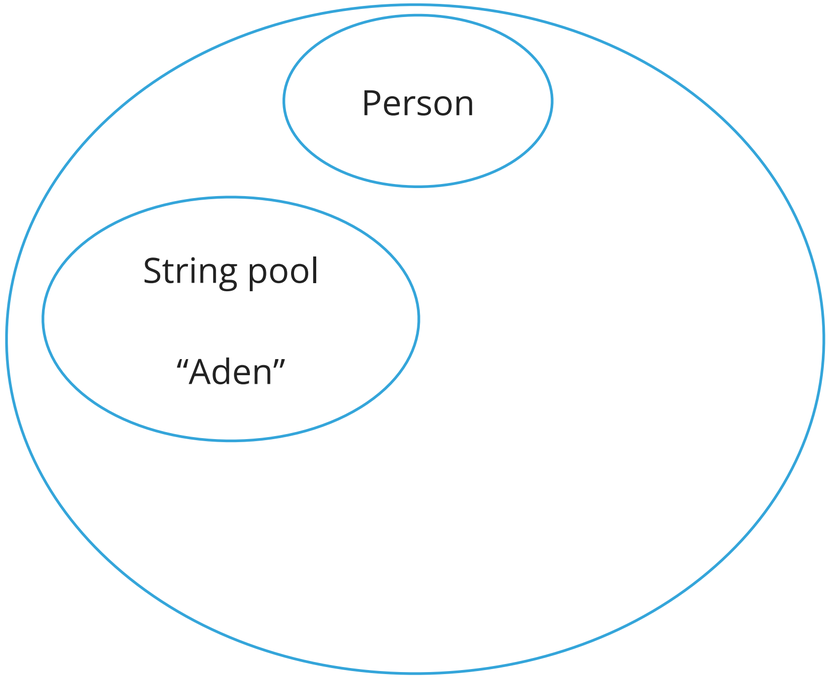



```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
        int number = 1234;
        String name = “Aden”;
        Person aden = null;
        aden = new Person(name, number)
  }
```

Once the constructor call ends, it’s wiped from the stack

# Stack vs heap in Java (walkthrough)



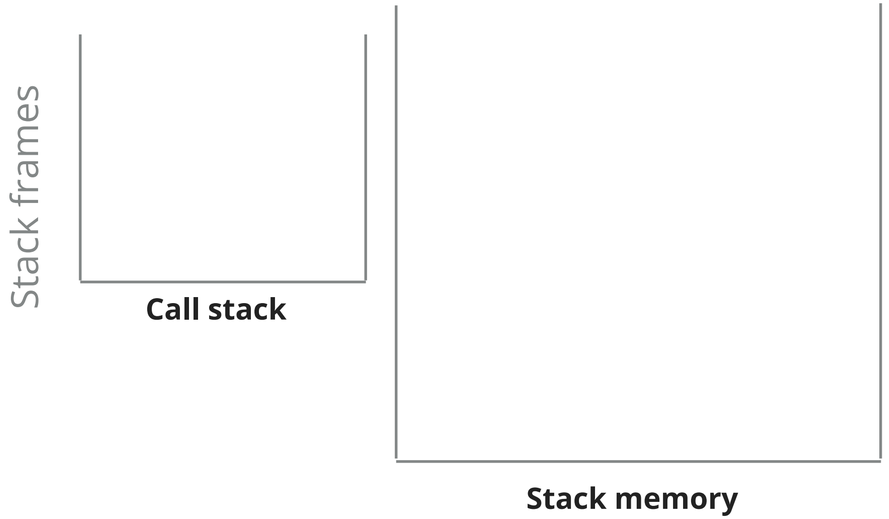



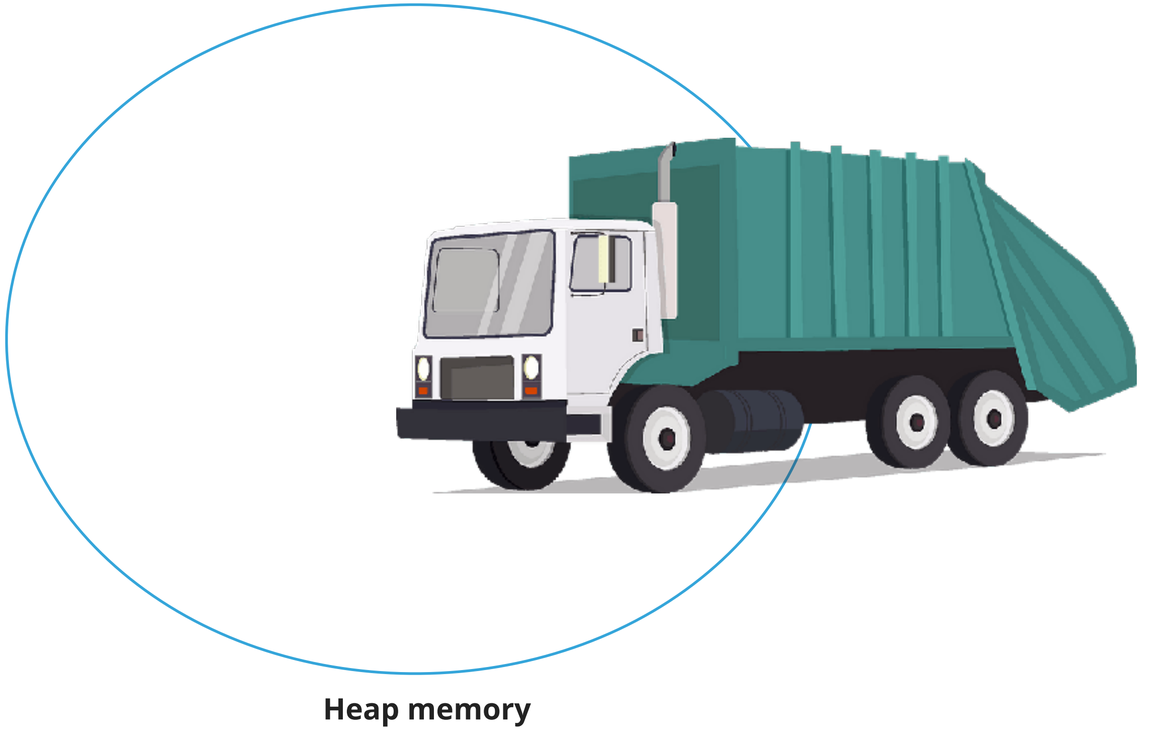

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    public static void main(String args[]) {
        int number = 1234;
        String name = “Aden”;
        Person aden = null;
        aden = new Person(name, number)
    }
```

We’re just left with our heap objects with no references to them :’(

Once the main call ends, it’s wiped from the stack

# Stack vs heap in Java (walkthrough)




```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
   }

   public static void main(String args[]) {
        int number = 1234;
        String name = “Aden”;
        Person aden = null;
        aden = new Person(name, number)
  }
```

Java automatically runs a garbage collector to get rid of heap objects that have been unreferenced and unused :D

# Summary

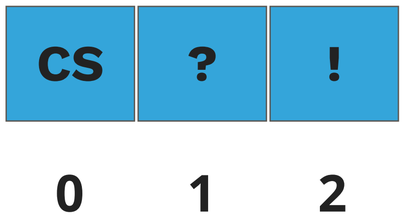
- The memory in the stack is fast and is for primitives & function calls
- The memory in the heap is slower and is for objects
- A garbage collector comes around and collects unused memory in the heap (as a programmer, you don’t have much control over this)

# Doubly Linked Lists (conceptually)

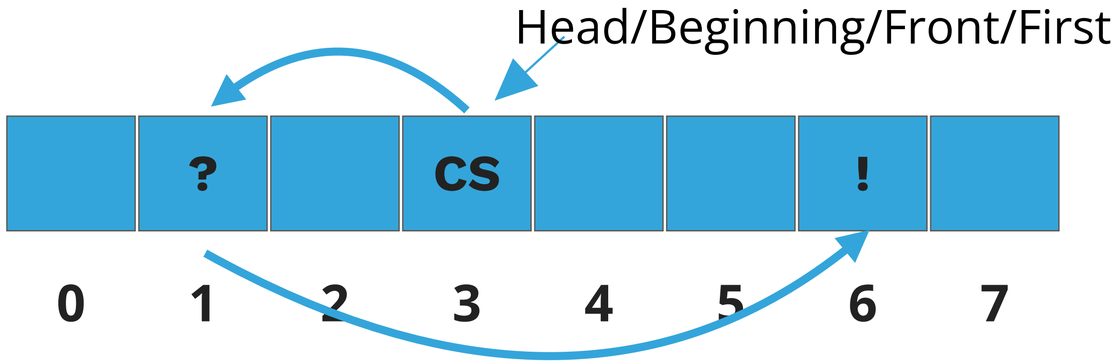
# Linked Lists

- Dynamic linear data structures.
- In contrast to sequential data structures, linked data structures use pointers/links/references from one object to another.
  - For example, the list of elements CS, ?, ! could be in very different memory locations. We just need a pointer to the head and links to subsequent elements to reconstruct it.

What the user's mental model is:




How it is in memory:



# One downside of SLLists

- Inserting at the back of a SLList is much slower than the front.

```
public void add(E element) {
    // Save the old node
    Node oldHead = head;

    // Make a new node and assign it to head. Fix pointers.
    head = new Node();
    head.element = element;
    head.next = oldHead;

    //increment size
    size++;
};
```

Q: What's the run time of each method?

A: O(1)

```
//adds to the tail of the list
public void addLast(E element) {

    Node finger = head;
    while (finger.next != null) {
        finger = finger.next;
    }

    Node n = new Node();
    n.element = element;
    finger.next = n;
    size++;
};
```

A: O(n)

# Change: why not a tail pointer?

Suppose we want to support `add`, `get`, and `remove` operations for both ends. Will having a `tail` pointer result for fast operations on long lists?

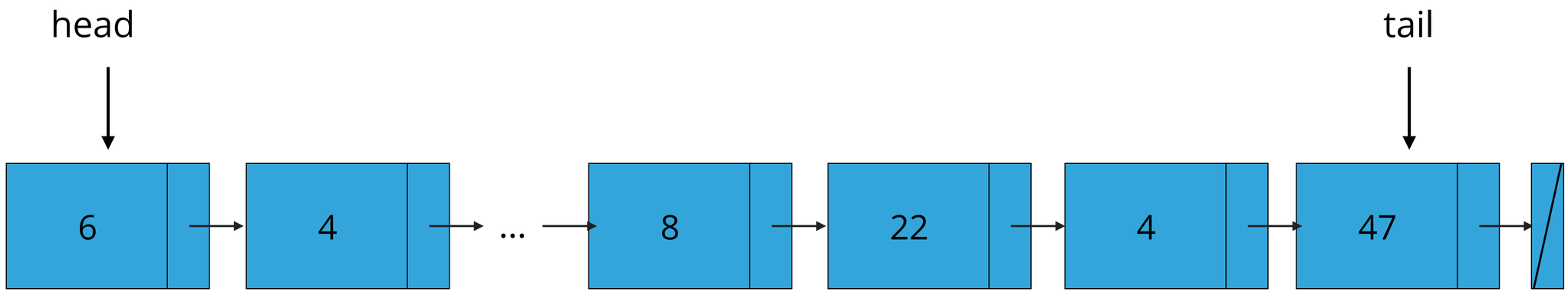
A. Yes

B. No, add would be slow. addLast()

C. No, get would be slow. getLast()

D. No, remove would be slow. removeLast()

slido.com #627



# A tail pointer is not enough

Suppose we want to support `add`, `get`, and `remove` operations for both ends. Will having a `tail` pointer result for fast operations on long lists?

A. Yes

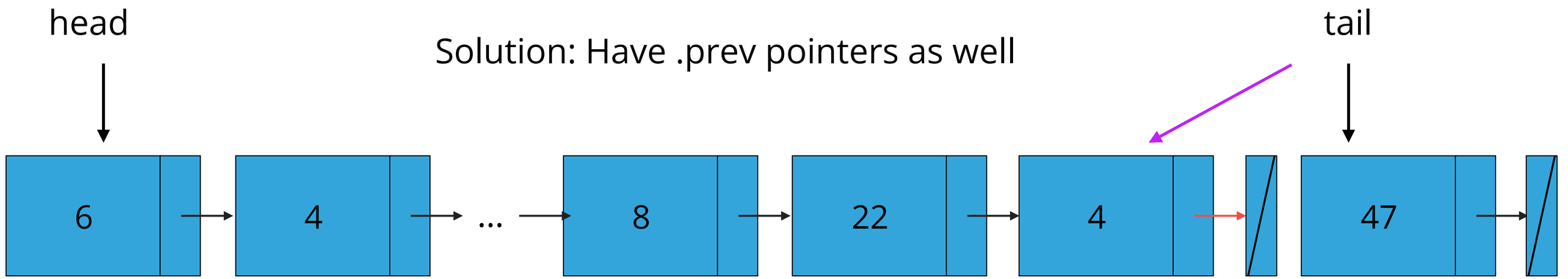
B. No, add would be slow.

C. No, get would be slow.

**D. No, remove would be slow.**

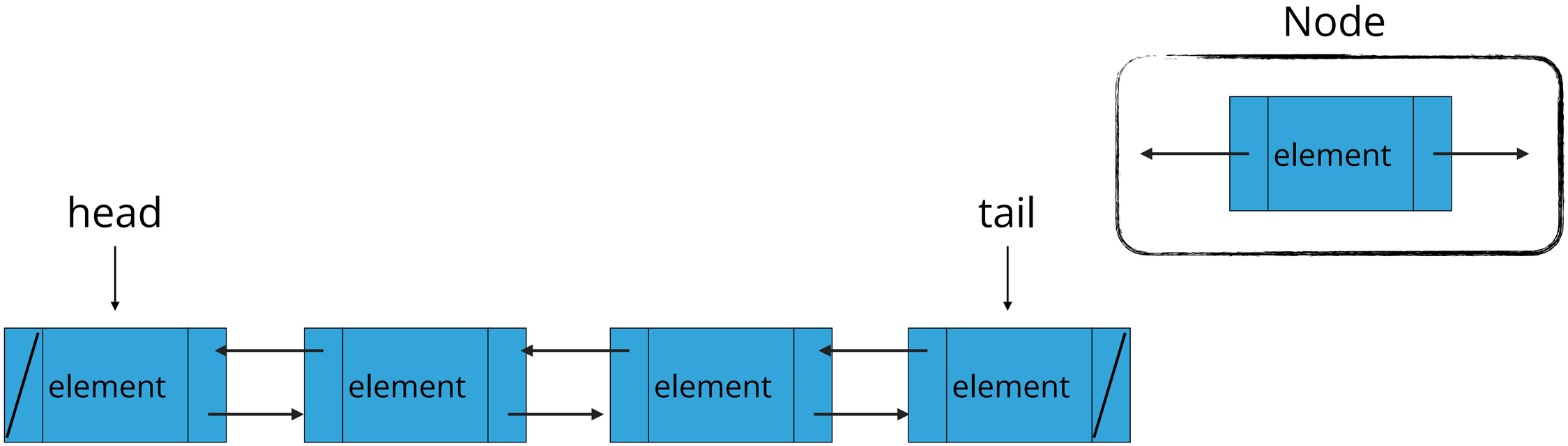
removeLast requires two actions:

- Setting 4's `next` variable to null.
- Setting `tail` equal to 4's memory location.
  - Have to search through almost the whole list to find the 4 node (second to last).



# Recursive Definition of Doubly Linked Lists

- A doubly linked list is either empty (null) or a node having a reference to a doubly linked list.
- Node: is a data type that holds any kind of data and **two references** to the **previous** and **next** node.



# DoublyLinkedList & Node

```
public class DoublyLinkedList<E> implements List<E> {
    private Node head; // head of the doubly linked list
    private Node tail; // tail of the doubly linked list
    private int size; // number of nodes in the doubly linked list

    /**
     * This nested class defines the nodes in the doubly linked list with a value
     * and pointers to the previous and next node they are connected.
     */
    private class Node {
        E element;
        Node next;
        Node prev;
    }
```

2 changes from SLL: add tail as instance variable, add prev pointer in Node class

# DLL walkthrough

# `DoublyLinkedList()`: **Constructs an empty DLL**

head

tail

size

What should happen?

```
DoublyLinkedList<String> dll = new DoublyLinkedList<String>();
```

# `DoublyLinkedList()`: **Constructs an empty DLL**

```
DoublyLinkedList<String> dll = new DoublyLinkedList<String>();
```

head = null

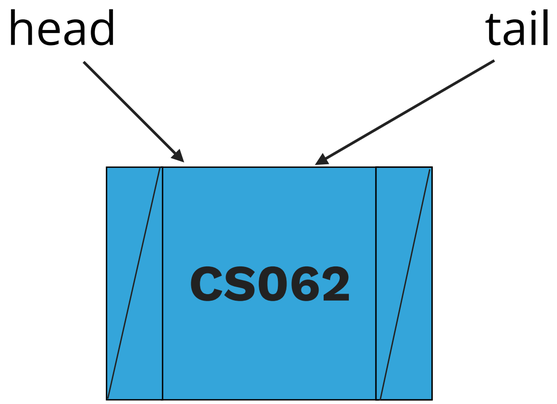
tail = null

size = 0

What should happen?

```
dll.add("CS062");
```

`add(E element)`: **Inserts the specified element at the head of the doubly linked list.**

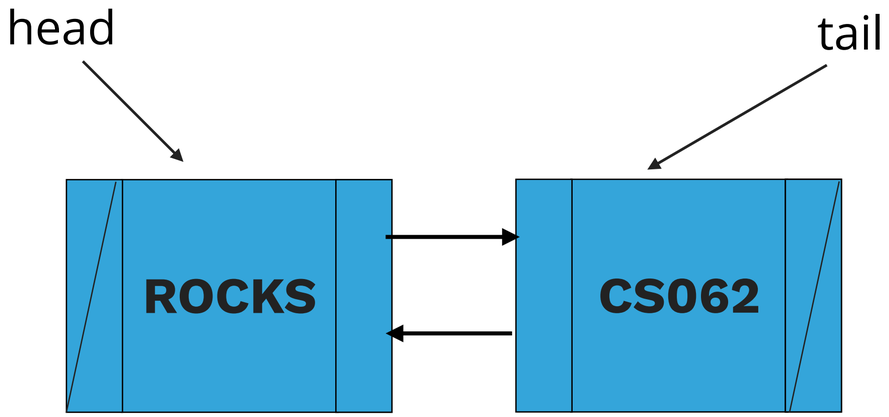



```
dll.add("CS062")

size=1
```

What should happen?

```
dll.addFirst("ROCKS");
```

# `addFirst(E element)`: **Inserts the specified element at the head of the doubly linked list**

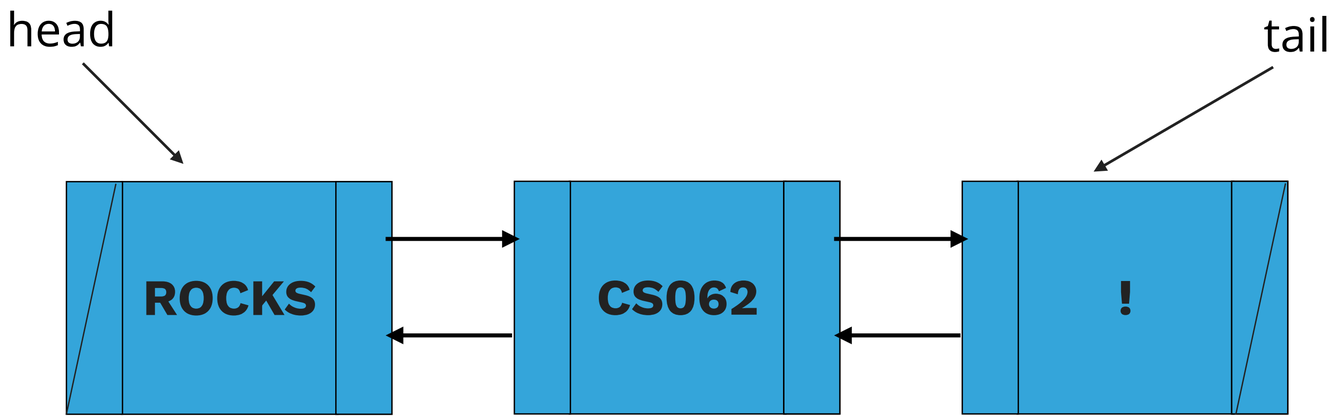



```
dll.addFirst("ROCKS")

size=2
```

What should happen?

```
dll.addLast("!");
```

# `addLast(E element)`: **Inserts the specified element at the tail of the doubly linked list**




```
dll.addLast(“!”)

size=3
```

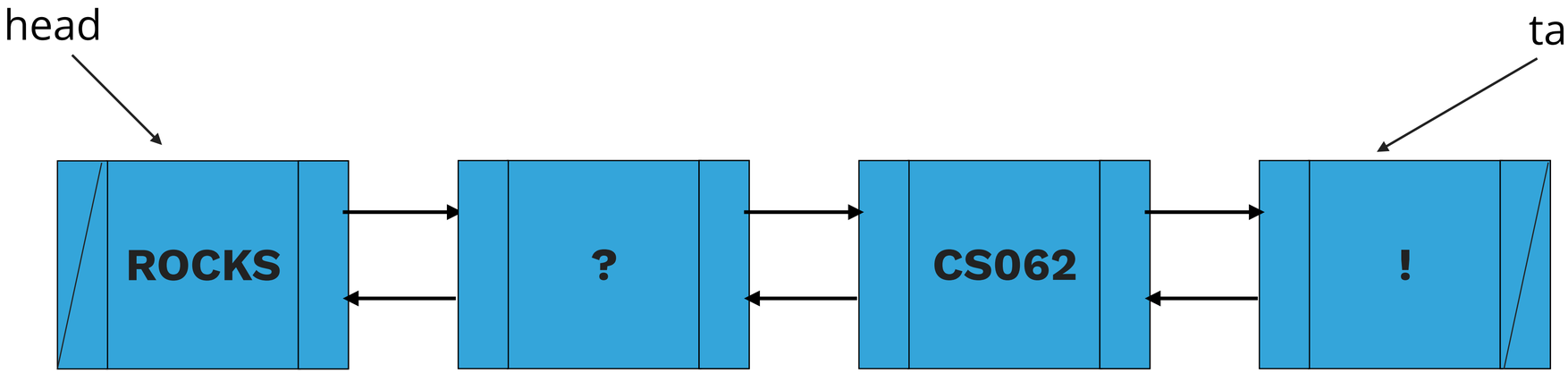
What should happen?

```
dll.add(1,“?”);
```

# `add(int index, E element)`: **Adds element at the specified index**

`dll.add(1,“?”)`

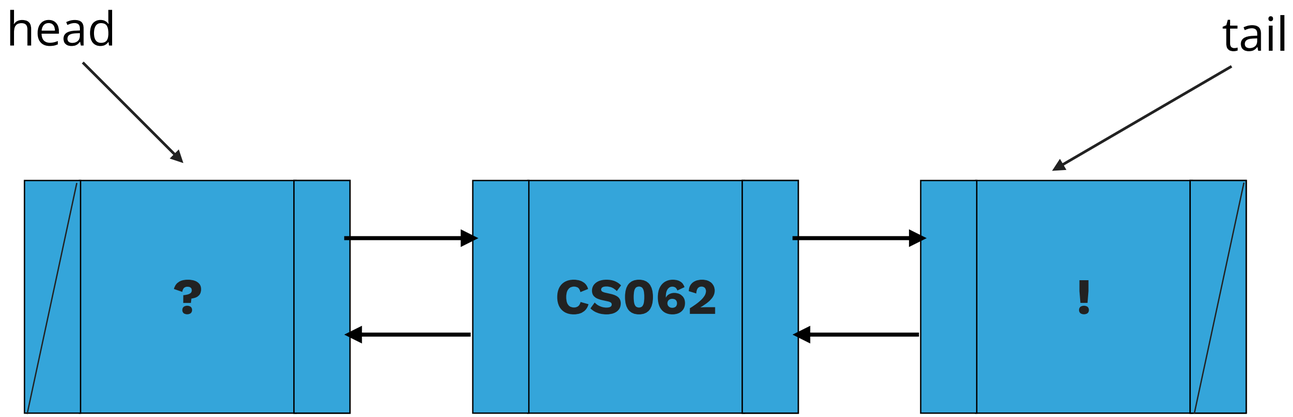
`size=4`




What should happen?

```
dll.remove();
```

# `remove()`: Removes and returns the head of the doubly linked list

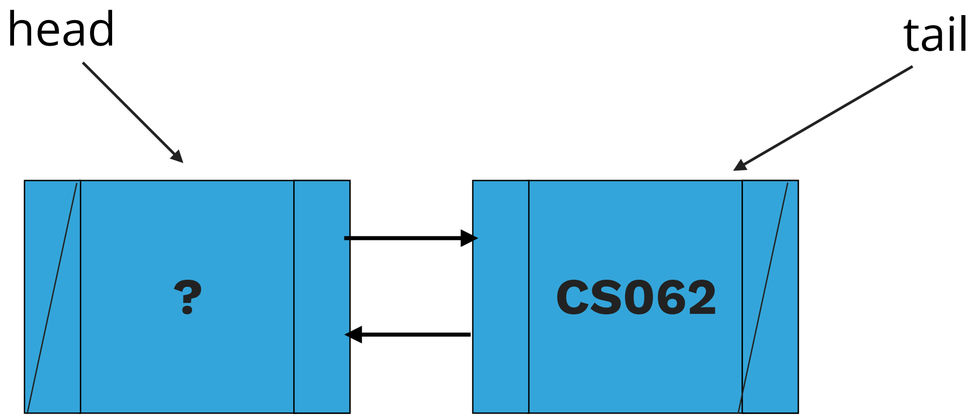



```
dll.remove()

size=3
```

What should happen?

```
dll.removeLast();
```

# `removeLast()`: Removes and returns the tail of the doubly linked list

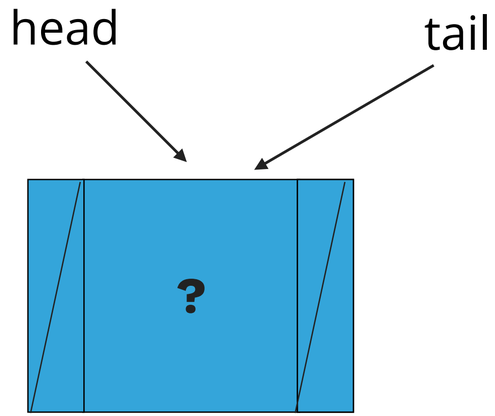



```
dll.removeLast()

size=2
```

What should happen?

```
dll.remove(1);
```

`remove(int index)`: **Removes and returns the element at the specified index**




```
dll.remove(1)

size=1
```

# *Worksheet time!*

- Do Qs 1-2 on your worksheet.

Suppose x and t are references to different Nodes in a doubly linked list. What is the effect of the following code fragment?

```
t.prev = x;
t.next = x.next;
x.next.prev = t;
x.next = t;
```

What if the code was in a different order?

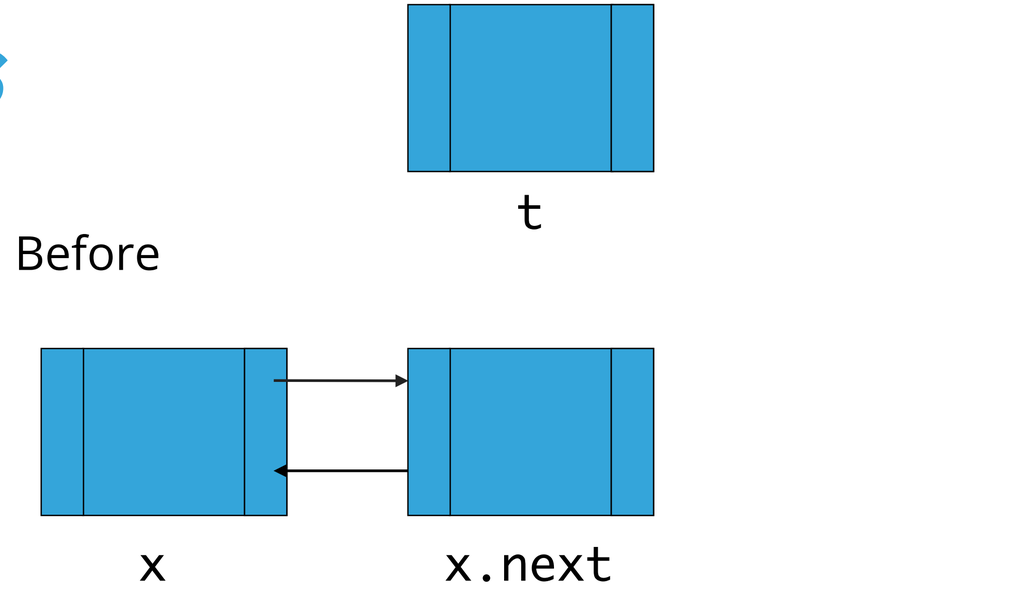
```
x.next = t;
x.next.prev = t;
t.next = x.next;
t.prev = x;
```

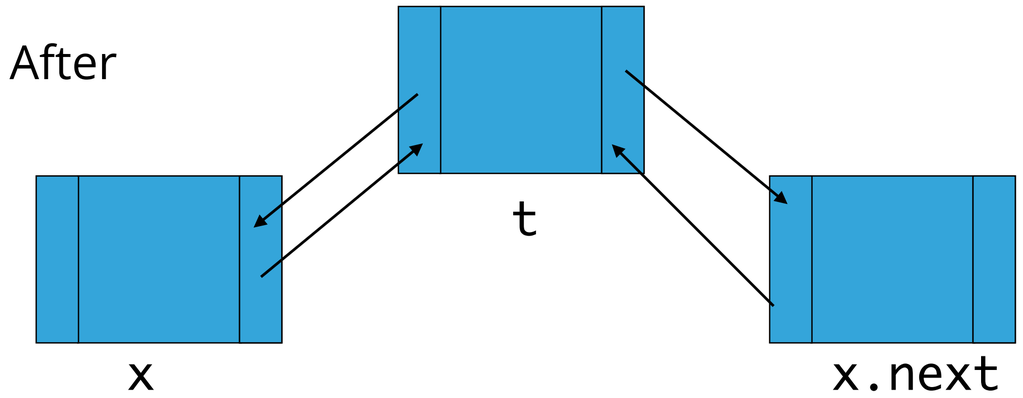
# Worksheet answers

Suppose x and t are references to different Nodes in a doubly linked list. What is the effect of the following code fragment?

```
t.prev = x;
t.next = x.next;
x.next.prev = t;
x.next = t;
```

**It inserts t between x and x.next.**



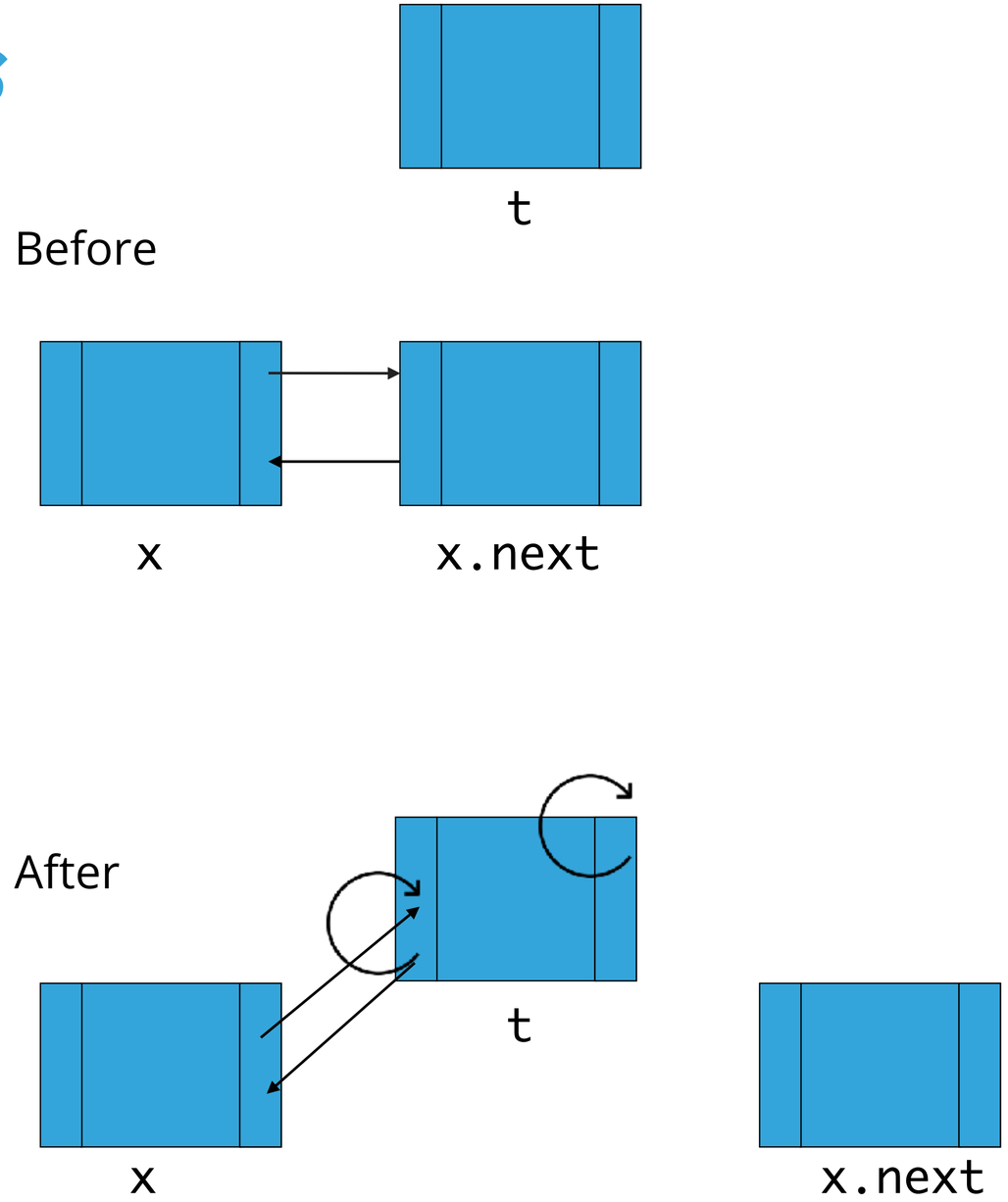



# Worksheet answers

What if the code was in a different order?

```
x.next = t;
x.next.prev = t;
t.next = x.next;
t.prev = x;
```

**It's gibberish**



# More Implementation Adding

# Adding to the head of a DLL

Adding to head of SLL

```
//adds to the head of the list
public void add(E element) {
    // Save the old node
    Node oldHead = head;

    // Make a new node and assign it to head. Fix pointers.
    head = new Node();
    head.element = element;
    head.next = oldHead;

    //increment size
    size++;
};
```

Largely, we can reuse SLL code, but we need to think about what to do with .prev pointers and the tail pointer.

There are 2 cases to adding:

1. The DLL is empty, so we have created the first element, and need to set tail = head
2. The DLL has an element, so tail is set properly. But then we need to find the old head, and set oldHead.prev = new node.

# Adding to the head of a DLL

```
/**
 * Inserts the specified element at the head of the doubly linked list.
 *
 * @param element
 *              the element to be inserted
 */
public void addFirst(E element) {
    // Save the old node
    Node oldHead = head;

    // Make a new node and assign it to head. Fix pointers.
    head = new Node();
    head.element = element;
    head.next = oldHead;
    head.prev = null;

    // if first node to be added, adjust tail to it.
    if (tail == null) {
        tail = head;
    } else {
        oldHead.prev = head;
    }
    size++; // increase number of nodes in doubly linked list.
}
```

Largely, we can reuse SLL code, but we need to think about what to do with .prev pointers and the tail pointer.

There are 2 cases to adding:

1. The DLL is empty, so we have created the first element, and need to set tail = head

2. The DLL has an element, so tail is set properly. But then we need to find the old head, and set oldHead.prev = new node.

# *Worksheet time!*

- Insert element at the tail for doubly linked lists.

```
public void addLast(E element) {
    // Save old tail


    // Make a new node and assign it to tail. Fix pointers.









    // if first node to be added, adjust head to it.


    // else fix next pointer to tail


    // increase size


}
```

# Worksheet answers

```
public void addLast(E element) {
    // Save the old node
    Node oldTail = tail;

    // Make a new node and assign it to tail. Fix pointers.
    tail = new Node();
    tail.element = element;
    tail.next = null;
    tail.prev = oldTail;

    // if first node to be added, adjust head to it.
    if (head == null) {
        head = tail;
    } else {
        oldTail.next = tail;
    }
    size++;
}
```

# More Implementation Removing

# Review: Retrieve and remove head (SLL)

```java
 /**
 * Removes and returns the head of the singly linked list.
 *
 * @return the head of the singly linked list.
 */
public E remove() {
   // Make a temporary pointer to head
   Node temp = head;
   // Move head one to the right
   head = head.next;
   // Decrease number of nodes
   size--;
   // Return element held in the temporary pointer
   return temp.element;
}
```

Just advance the head pointer to head.next

# Retrieve and remove head (DLL)

```java
/**
 * Removes and returns the head of the doubly linked list.
 *
 * @return the head of the doubly linked list.
 */
public E removeFirst() {
  // Create a pointer to head
  Node temp = head;
  // Move head to next
  head = head.next;
  // if there was only one node in the doubly linked list
  if (head == null) {
    tail = null
  } else {
        head.prev = null;
  }
  // decrease number of nodes
  size--;
  // return old head's element
  return temp.element;
}
```

only this if statement block is different

Think of all the possible cases.
1. There is only 1 node and now the DLL is empty
2. There is more than 1 node, so tail doesn't need to change.

1. Set tail to null, since the DLL is empty now
2. head.prev = null to clean up prev pointers

# Retrieve and remove last element (DLL)

This should be in O(1), as this is what we motivated in the start of lecture.

O(1) means just pointer manipulation in your implementation - no loops!

```java
/**
 * Retrieves and removes the tail of the doubly linked list.
 *
 * @return the tail of the doubly linked list.
 */
public E removeLast() {

    Node temp = tail;
    tail = tail.prev;

    // if there was only one node in the doubly linked list.
    if (tail == null) {
        head = null;
    } else {
        tail.next = null;
    }
    size--;
    return temp.element;
}
```

# Review: Retrieve and remove element from a specific index (SLL)

```
public E remove(int index) {
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
    if (index == 0) {
        return remove();
    } else {
        Node previous = null;
        Node finger = head;
        // search for value indexed, keep track of previous
        for(int i=0; i<index; i++){
            previous = finger;
            finger = finger.next;
        }
        previous.next = finger.next;

        size--;
        // finger's value is old value, return it
        return finger.element;
    }
}
```

use fingering method to find the right node

skip the pointer

# Worksheet time!

Fill in the blanks to implement remove at a specific index for DLLs

```
public E remove(int index) {
        if (index >= size || index < 0) {
            throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
        }
        if (index == 0) {
            return _________________
        } else if (index == size - 1) {
            return _________________
        } else {
            Node previous = null;
            Node finger = head;
            // search for value indexed, keep track of previous
            while (index > 0) {
                previous = _________
                finger = ___________
                ___________ // what do we do with index?
            }
           // found the element to remove, change pointers
            previous.next = ____________;
            _____________ = previous;
            size--;
            return finger.element;
        }
    }
```

# Worksheet answers!

Fill in the blanks to implement remove at a specific index for DLLs

```
public E remove(int index) {
        if (index >= size || index < 0) {
            throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
        }
        if (index == 0) {
            return _________________
        } else if (index == size - 1) {
            return _________________
        } else {
            Node previous = null;
            Node finger = head;
            // search for value indexed, keep track of previous
            while (index > 0) {
                previous = _________
                finger = ___________
                ___________ // what do we do with index?
            }
           // found the element to remove, change pointers
            previous.next = ____________;
            _____________ = previous;
            size--;
            return finger.element;
        }
    }
```

removeFirst();

removeLast();

finger;

finger.next;

index--;

finger.next

finger.next.prev

# *Worksheet time!*

Check all the methods that are the same between singly linked lists and doubly linked lists (i.e., their implementation is the same, you can just copy/paste the SLL code and it will work for DLL).

- ☐ void clear()
- ☐ E get(int index)
- ☐ boolean isEmpty()
- ☐ E set(int index, E element)
- ☐ int size()

# *Worksheet answers!*

Check all the methods that are the same between singly linked lists and doubly linked lists (i.e., their implementation is the same, you can just copy/paste the SLL code and it will work for DLL).

- ☐ void clear() — no, need to also set tail = null
- ☐ E get(int index) — mostly — could do if index == size-1, return tail.element to speed up, not required
- ☐ boolean isEmpty() — yes, if size == 0 implementation
- ☐ E set(int index, E element) — yes, just replacing element means no pointer manipulation required
- ☐ int size() — yes

# Linked List invariants

- The head variable will always point to the start of the list or null
- The tail variable will always point to the end of the (doubly linked) list or null
- The size variable is always the total number of nodes

# Running times

| operation | SLL time complexity | DLL time complexity | explanation |
|---|---|---|---|
| addFirst | O(1) | O(1) | pointer manip |
| addLast | O(n) | O(1) | SLL doesn't have tail pointer |
| add(index) | O(n) | O(n) | iterate through whole list |
| removeFirst | O(1) | O(1) | pointer manip |
| removeLast | O(n) | O(1) | SLL doesn't have tail pointer |
| remove(index) | O(n) | O(n) | iterate through whole list |
| get(index) | O(n) | O(n) | iterate through whole list |
| set(index) | O(n) | O(n) | iterate through whole list |
| clear() | O(1) | O(1) | pointer manip |

# What about ArrayLists?

| operation | SLL time complexity | DLL time complexity | ArrayList | Explanation |
|---|---|---|---|---|
| addFirst | O(1) | O(1) | O(n) | Need to shift rest of array |
| addLast | O(n) | O(1) | O(1)+ | Could call resize() |
| add(index) | O(n) | O(n) | O(n) | Need to copy/shift elements |
| removeFirst | O(1) | O(1) | O(n) | Need to shift rest of array |
| removeLast | O(n) | O(1) | O(1)+ | Could call resize() |
| remove(index) | O(n) | O(n) | O(n) | Need to copy/shift elements |
| get(index) | O(n) | O(n) | **O(1)** | Just array indexing |
| set(index) | O(n) | O(n) | **O(1)** | Just array indexing |
| clear() | O(1) | O(1) | O(n) | Loop through array to set = null |

# Space complexity?

- Space complexity is a measure of the total memory an algorithm/data structure uses.
- Both linked lists and ArrayLists use O(n) space complexity (where n is the number of items).
- However, linked lists are slightly higher, since there is additional memory overhead since each element is a Node with pointers. In an Arraylist, each element is just the element itself.

# Run time summary

- If we are iterating through the linked list to find a specific Node (“finger”), O(n) worst case
  - get(), set(), add() with index, remove() with index
- If we are just moving pointers around, O(1) worst case
  - add() from the head, remove() from the head, clear()

- ArrayLists benefit from O(1) get() and set() times, so often are used over linked lists in practice.

# Lecture 7 wrap-up

- DLLs maintain prev pointers and a reference to the end of the list.
- Lab tonight: learn how to use the Java debugger and get out of a maze of linked list pointers. (New assignment!)
- Part II of Darwin (everything else) due next Tues 11:59pm
- Checkpoint I is 9/29. If you have SDRC accommodations, please schedule them **now**. We cannot offer alternate proctoring in-class (e.g., if you have extra time, please get it proctored via the SDRC). 1 double sided handwritten cheat sheet OK.
  - Lab next week is checkpoint 1 review, includes next week's 2 lectures

# Resources

- Linked lists from the textbook: https://algs4.cs.princeton.edu/13stacks/
- See slides following this for one more practice problem

# Bonus practice problem

- Add a method `removeAfter(Node node)` in the DoublyLinkedList class that removes the node following the given one.

# Bonus answer

```
public void removeAfter(Node node) {
    if (isEmpty() || node == null) {
        return;
    }
    for (Node current = head; current != null; current = current.next) {
        if (current == node) {
            if (current.next != null) {
                current.next = current.next.next;
                if(current.next != null){
                    current.next.prev = current;
                }
                size--;
            }
            break;
        }
    }
}
```