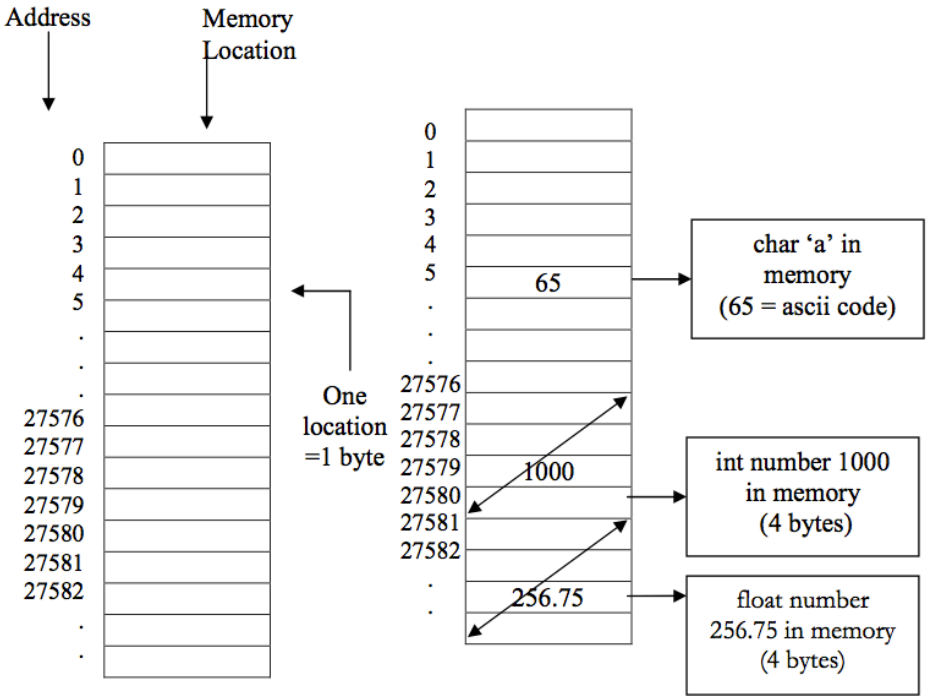
Assignment 2

Objects, arrays and linked lists in memory
Linked lists manipulation

Data Structures, Fall 2018
TA: Marija Stanojevic

Primitive types representation in memory

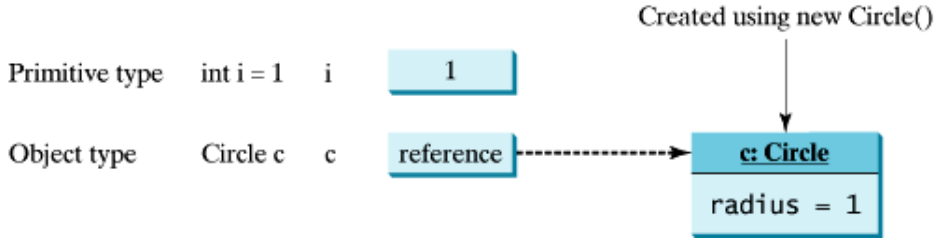


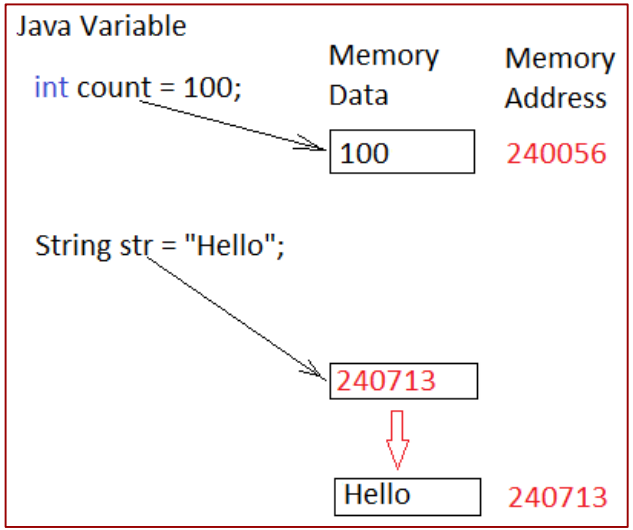
char - **1 byte (= 8 bits)**
short - 2 bytes (16-bits)
int - 4 bytes (32-bits)
long - 8 bytes (64-bits)
float - 4 bytes (32-bits)
double - 8 bytes (64-bits)
No cast from smaller to bigger or when equal (int -> float, short->long)
Need to cast from bigger to smaller (e. g. long -> short, double -> int)

[1] https://tex.stackexchange.com/questions/148778/how-to-draw-vertical-arrays-with-diagonal-arrows-to-represent-memory-in-tikz

Primitive vs Object types in memory

- When object is created (type starts with uppercase) two things are saved in memory:
  - **Values** of data fields (from previous slide)
  - **Reference** - address of the place where data is saved
- When primitive types (start with lowercase letters) instances are created, **only values** are saved in memory
- **Primitive types are:** int, short, double, float, long
- **Objects:** String, Circle, Node, LinkedList, Stack,...





What is the size of reference in memory

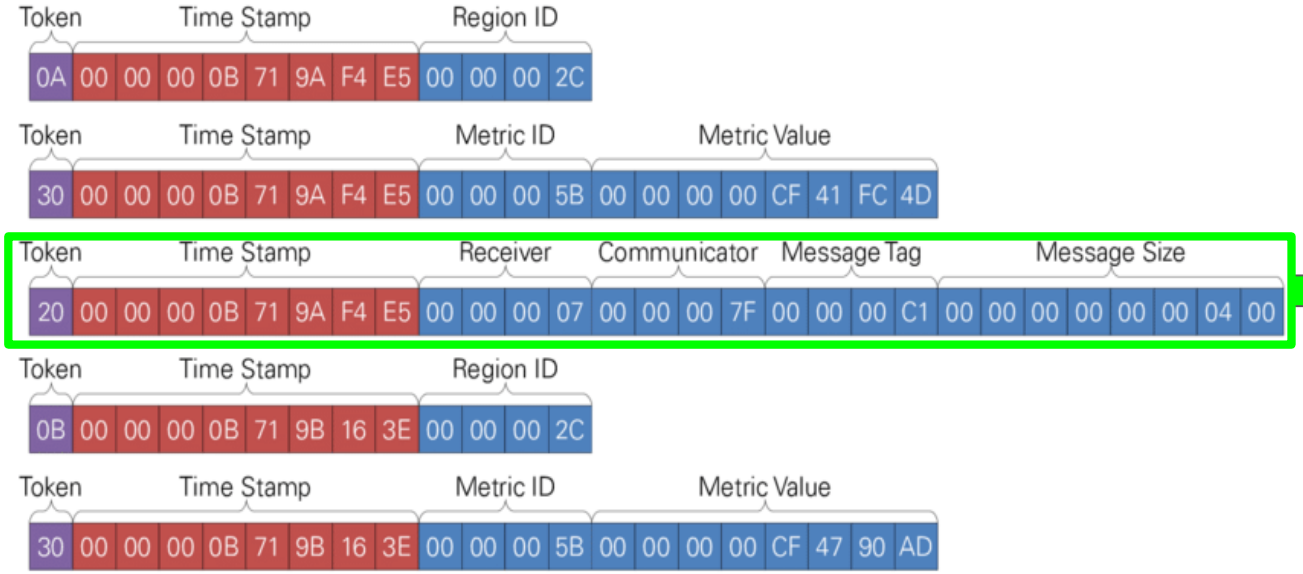
What is the difference between 32-bit and 64-bit operating systems?

Addresses / references storage require 32-bits (4 bytes) in older 32-bit operating systems and 64-bits (8 bytes) in new operating systems.

Since we can store address in 64-bits, that means that we can address memory of size 2^{64} bytes, because 1 row has one byte and we have 2^{64} combinations to address a row.

In old computers, we could address only 2^{32} bytes = $2^{2} * 2^{30}$ = 4 * 1GB = 4GB of memory. In new computers, we can address 2^{64} bytes = $2^{4} * 2^{60}$ = 4 * 1EB = 16EB

Objects representation in memory



Object Message takes 29 bytes to store data fields + 8 bytes to store reference to Message object

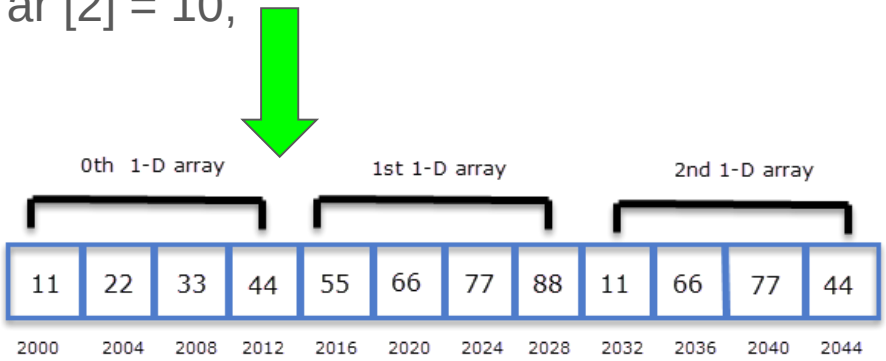
```
public class Message {
        private char token;
        private Date
timeStamp;
        private int receiver;
        private int
communicator;
        private int msgTag;
        private double
msgSize;

        // constructors
        // getters & setters
```

TimeStamp is reference to object Date, so it takes 8 bytes. Once timeStamp is initialized, additional 32 bytes are used to store its values

[2] https://www.researchgate.net/figure/represents-the-memory-representation-of-such-an-event-sequence-in-a-binary-encoding-All_fig21_284440851

Arrays representation in memory

```
int [] ar = new int [3];
ar[0] = 5;
ar[1] = 4;
ar [2] = 10;
```



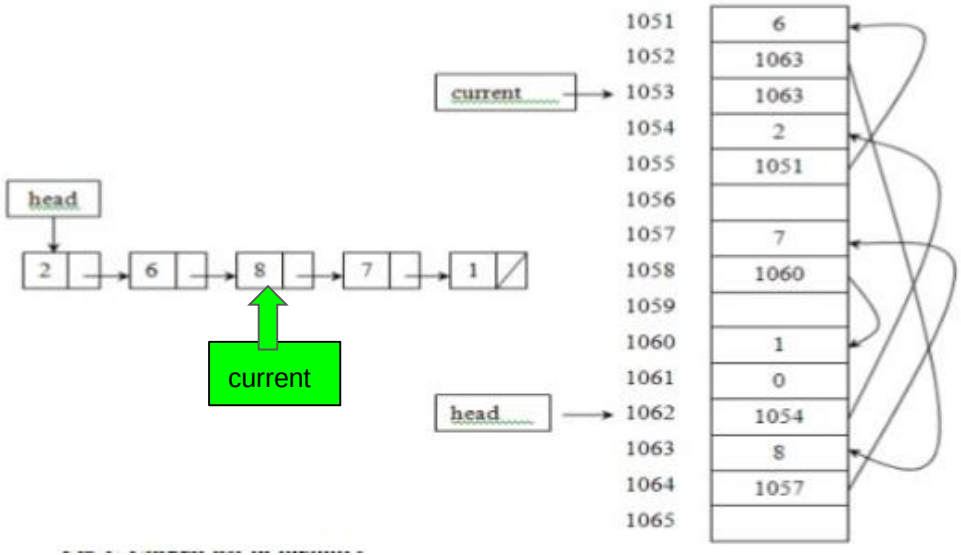
Arrays of objects would be of size **N * k**, where N is length of array and k is size of object in memory

Array of 10 messages takes 290 bytes to store data fields and 80 bytes to store references to the objects, so 370 bytes in total.

Message [] ar = new Message[10];

[3] https://overiq.com/c-programming/101/pointers-and-2-d-arrays/

Linked lists in memory



Head is a reference that shows address of first node (1054).

In address 1054, we have value 2 and in next address reference to second node (1051).

In address 1051, we have value 6 and in next address reference to third node (1063).

If we define new reference to third node (current), it would also contain value 1063, ie. address of third node


Linked Lists
HackerRank

Linked list - cases

Make sure your functions work for:

- Empty list (head = null)
- List with only one element (head.next = null)
- List with few elements
- If index/position is given as argument make sure your function works for:
  - negative index
  - index 0
  - index that is lower than number of elements in the list
  - index that is equal to the number of elements in the list
  - index that is equal to number of elements in the list + 1
  - index that is bigger than number of elements in the list + 1

Class Node

```
private class Node {
        private int data;
        private Node next;
        private Node(int data) {
                this.data =
data;
                next = null;
    }
    private Node(int data, Node
next) {
                this.data =
data;
                this.next =
next;
```

Node initialization:

```
    Node x = null;
    Node a = new Node(5);
    Node b = new Node(10, null);
    Node c = new Node(15, new
Node(3));
```

When variable is just declared (e.g. x), just it's reference is in memory, ie. 8 bytes.

When variable is declared and initialized (e.g. a, b, c), both reference and data are stored in memory: 8 bytes for reference + 12 bytes for Node data.

x = a; - copies reference/address of Node with value 5 into x, so now x and a both reference that Node.

Go through the list

```
private Node head;
public void goThroughTheList(int item) {
        Node current = head;
        count = 0;
        sum = 0;
        belongs = false;
        while (current != null) {
                count ++;
                sum += current.data;
                current = current.next;
                if (current.data ==
item) {
                        belongs = true;
        }
    }
}
```

- Current.next in while condition means that last element is not accessed.
- Don’t use it, except to add/remove element from the last position

Add element at index

```
private Node head;
public boolean add (int item, int index)
{
        if (index < 0 || index > size())
{
                return false;
    }
    if (index == 0) {
        head = new Node(item, head);
        return true;
    }
    Node current = head;
    for (int i = 0; i < index-1; i++) {
        current = current.next;
    }
    current.next = new Node(item,
current.next);
    return true;
}
```

Remove element by value

```
private Node head;
public boolean removeByValue(int item) {
          if (head == null) {
          return false;
     }
     if (head.data == item) {
          head = head.next;
          return true;
     }
     Node current = head.next;
          Node prev = head;
     while (current != null) {
          if (current.data == item) {
                    prev.next = current.next;
                    return true;
          }
          prev = current;
          current = current.next;
     }
     return false;
}
```

Remove element by index

```
private Node head;
public boolean removeByIndex (int item, int
index) {
          if (index < 0 || index >= size())
{
                    return false;
     }
     if (index == 0) {
          head = head.next;
          return true;
     }
     Node current = head;
     for (int i = 0; i < index - 1; i++) {
          current = current.next;
     }
     current.next = current.next.next;
     return true;
}
```

java.api LinkedList

LinkedList is generic class. Data part of the nodes can be of any type.

add(E e) - adds to the end of the list
add(int index, E e) - adds to the position
addFirst(E e) - adds to the beginning
addLast(E e) - adds to the end of the list

clear() - removes all elements from the list

get(int index) - gets value at position
getFirst() - gets value from head
getLast() - gets value of the last node

remove() - removes first element of the list
remove(int index) - removes from position
removeFirst() - removes first element
removeLast() - removes last element

size() - returns size of the list