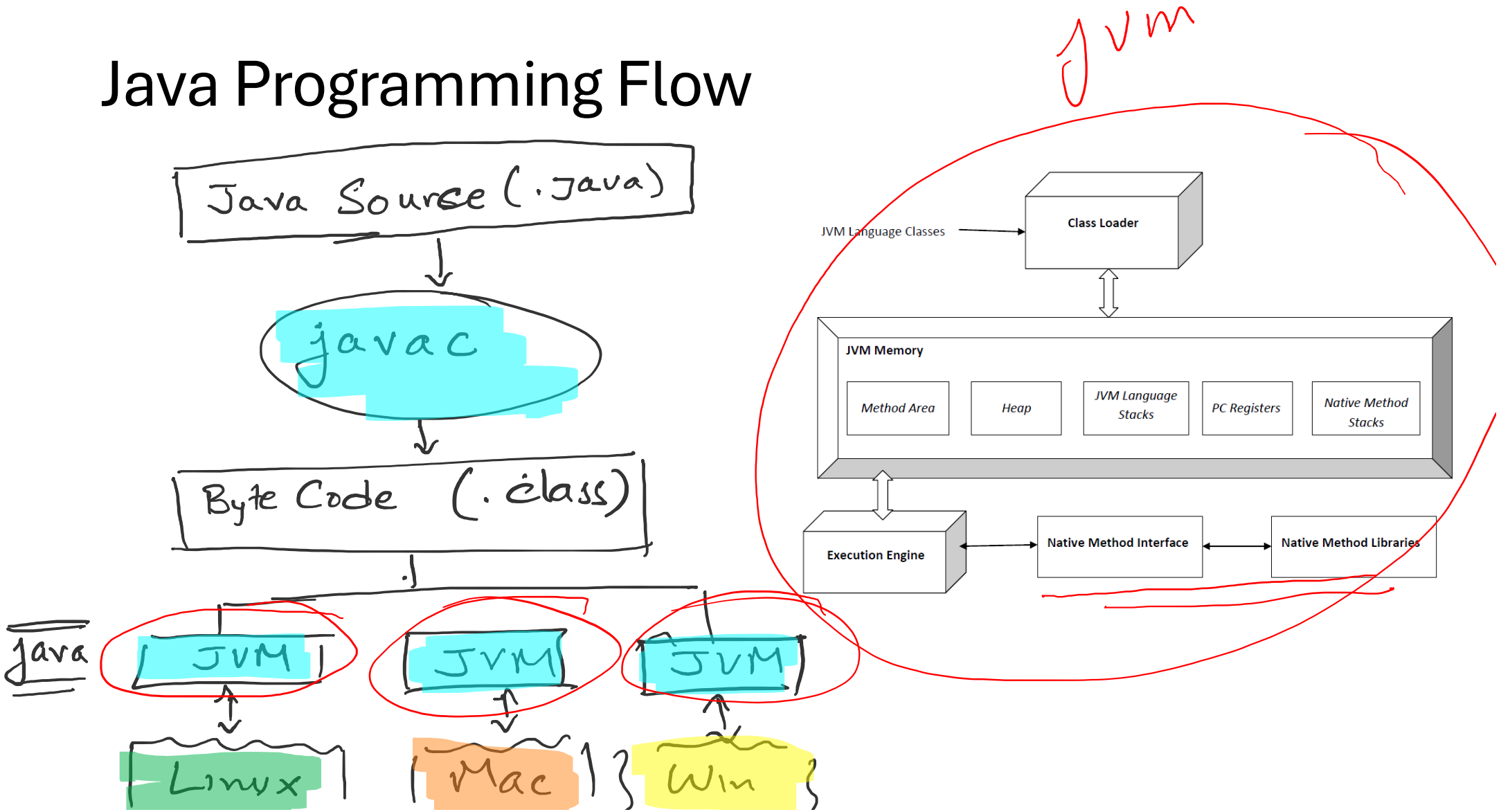
COL106 - Data Structures and Algorithms

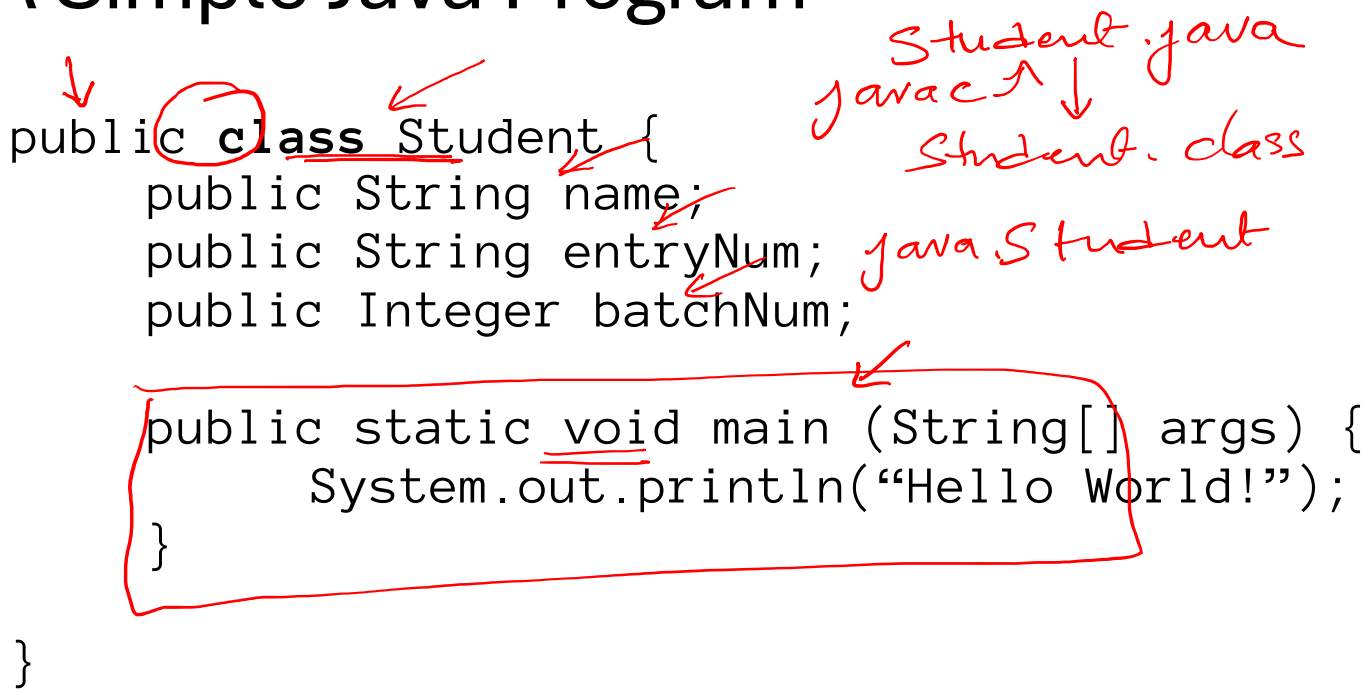
Java Programming Language

- An object-oriented, high-level language with automatic garbage collection that follows “Write-Once Run Anywhere” (WORA) paradigm
  - It is not an interpreted language, but is compiled into an intermediate byte-code – **javac**
  - Executed over a “virtual machine” that mediates with the underlying hardware and the byte-code – **java**

Java Programming Flow



A Simple Java Program



Java – Instantiating Objects

```
public class Student {
    public String name;
    public String entryNum;
    public Integer batchNum;

    public static void main (String[] args) {
        System.out.println("Hello World!");

        st1 = new Student();
        st1.name = "Veeru";
        st2 = new Student();
        st2.name = "Basanti";
    }
}
```

Goals of Object-oriented Approach

- Robustness
  - We want software to be capable of handling unexpected inputs that are not explicitly defined for its application.
- Adaptability
  - Software needs to be able to evolve over time in response to changing conditions in its environment.
- Reusability
  - The same code should be usable as a component of different systems in various applications.

Core Ideas of Object-Oriented Approach

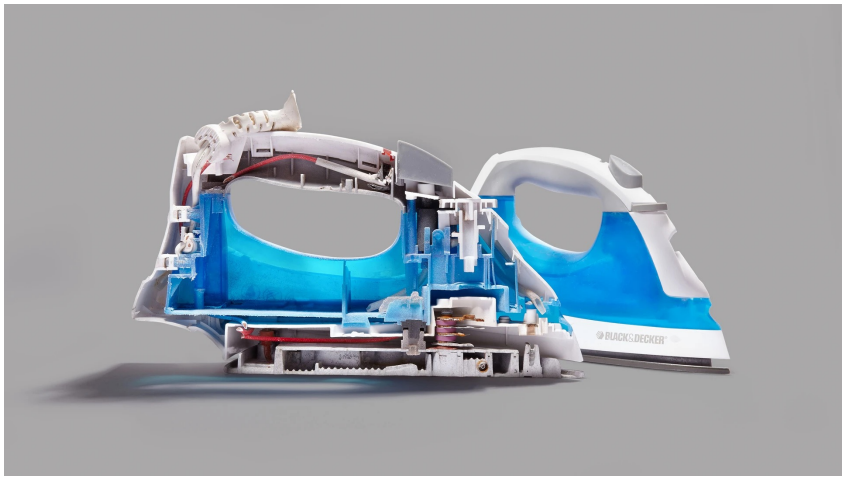
- **Idea 1 : Abstraction**
  - We only need to know ”what” needs to be the behavior, not “how” it is implemented
  - Abstraction is to distill a system to its most fundamental parts.

- Applying the abstraction paradigm to the design of data structures gives rise to abstract data types (ADTs).

- Abstract Data Types (ADTs) are mathematical models of a data structure
  - Type of data stored
  - Operations supported on them and the parameters of these operations
- An ADT specifies **what** each operation does, but **not how** it does it.
- The collective set of behaviors supported by an ADT is its **public interface**.

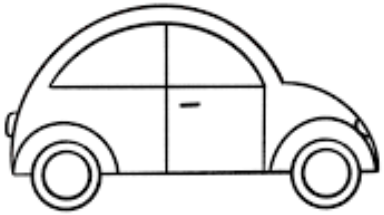
Core Ideas of Object-Oriented Approach

- **Idea 2 : Encapsulation**
  - Reveal only the necessary information, not the internal details of an object
    - Data as well as behavior
  - It is true of many every-day objects we use



https://www.fastcompany.com/90320298/these-oddly-satisfying-photos-reveal-the-inner-workings-of-everyday-objects

https://en.comun.app/blog/como-usar-cajeros-automaticos-de-manera-segura-consejos-y-trucos



https://thefairyglitchmother.com/car-drawing-for-kids-how-to-make-it-easy-peasy/

Core Ideas of Object-Oriented Approach

- **Idea 3 : Modularity**
  - A large software is divided into **separate** functional units
  - Each unit has a well-defined functionality (abstracted appropriately) so that different units can be independently be developed against their functional abstraction

Modular design of an iphone
(from trustedreviews.com)

The Course Structure

- Part I: Data-Structures as Implementations of Different Collection ADTs
  For each Collection type {
  - The Abstract Data Type (ADT) is introduced
  - Different data-structure implementations are discussed
  - Pros-Cons of the ADT and data-structure

  }
- Part II: Algorithms
  For each Algo concept {
  - Problem the algo is solving
  - Available approaches and choice of data-structure
  - Performance analysis

  }

Detailed lecture plans are on the website

All announcements and updates will be on the course website.

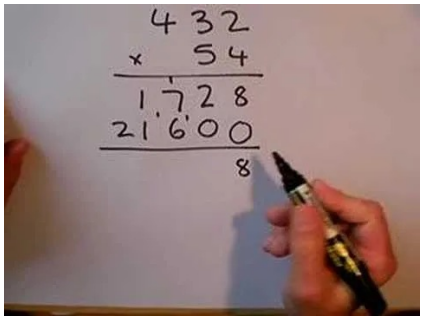
Slides will be uploaded at regular intervals (typically once a week)

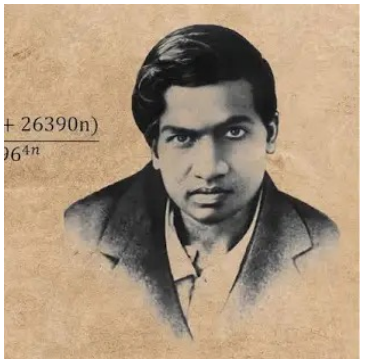
Abstract Data Type

- The key idea is that a (data)type is characterized by the operations you can perform on it.

- **Number** : you can compare, add, subtract etc.
- **String**: Concatenate and substring
- **Boolean** is something that you can negate …

- But there is no constraint on **how** any of these operations have to be done





Abstract Data Type of Bool

- Bool is an ADT that we would like to build

true: Bool
false: Bool

and: Bool x Bool → Bool
or: Bool x Bool → Bool
not: Bool → Bool

What are the ways in which we can implement Bool?

- A single bit 0 -> false, 1 -> true ?

Abstract Data Type of Bool

- Bool is an ADT that we would like to build

true: Bool
false: Bool

and: Bool x Bool → Bool
or: Bool x Bool → Bool
not: Bool → Bool

What are the ways in which we can implement Bool?

"true"
"false"

```
if ("true") {
    return "false"
}
if ("false") {
    return "true";
}
```

ADT Operator Classification

- **Creator**: create new objects of the type.

$t^* \rightarrow T$

- **Producer**: create new objects from old objects of the type

$T^+, t^* \rightarrow T$

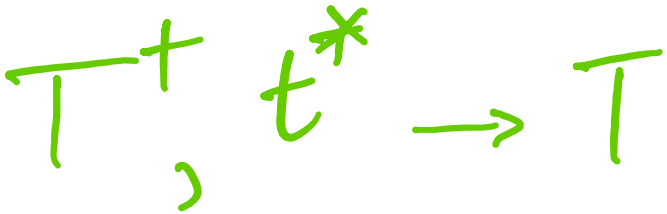
t : reference to another object (could be of different type)

T : reference to the object of the ADT we are working with

ADT Operator Classification

- **Observer**: take objects of the abstract type and return objects of a different type.

- **Mutator**: change objects' state / value.



Not all ADTs allow "mutator" operations

Mutable and Immutable ADTs

t : reference to another object (could be of different type)

T : reference to the object of the ADT we are working with

A SimpleString ADT

```
public abstract class SimpleString {  4 usages
    public char[] val;  1 usage

    public static SimpleString constructSimpleString(boolean flag) {  no usages
        SimpleString ret = new SimpleString();
        ret.val = flag ? new char[] {'t','r','u','e'} : new char[] {'f','a','l','s','e'};
        return ret;
    }

    public abstract SimpleString substr(int start, int end);  no usages

    public abstract int length();  no usages
    public abstract char charAt(int i);  no usages
}
```

SimpleString Implementation - 1

```
public class SimpleString {                                                    67  5
    public char[] val;  5 usages
    public static SimpleString constructSimpleString (boolean flag) {  1 usage
        SimpleString ret = new SimpleString();
        ret.val = flag ? new char[] {'t','r','u','e'} : new char[] {'f','a','l','s','e'};
        return ret;
    }
    public SimpleString substr (int start, int end) {  1 usage
        SimpleString ret = new SimpleString();
        ret.a = new char[end - start];
        System.arraycopy (this.val, start, ret.val,  destPos: 0,  length: end-start);
        return ret;
    }

    public int length () {  return val.length;  }  no usages
    public char charAt (int i) {  return val[i];  }  no usages

    public static void main (String[] args) {
        SimpleString ss1 = SimpleString.constructSimpleString ( flag: true); //ss1 will have 'true'
        SimpleString ss2 = ss1.substr (1,4); //ss2 will have 'rue';
    }
```

Creator

Observer

Observer

SimpleString Implementation - 1

```
public class SimpleString {                                                    67  5
    public char[] val;  5 usages
    public static SimpleString constructSimpleString (boolean flag) {  1 usage
        SimpleString ret = new SimpleString();
        ret.val = flag ? new char[] {'t','r','u','e'} : new char[] {'f','a','l','s','e'};
        return ret;
    }
    public SimpleString substr (int start, int end) {  1 usage
        SimpleString ret = new SimpleString();
        ret.a = new char[end - start];
        System.arraycopy (this.val, start, ret.val,  destPos: 0,  length: end-start);
        return ret;
    }

    public int length () {  return val.length;   }  no usages
    public char charAt (int i) {    return val[i];    }  no usages

    public static void main (String[] args) {
        SimpleString ss1 = SimpleString.constructSimpleString ( flag: true); //ss1 will have 'true'
        SimpleString ss2 = ss1.substr (1,4); //ss2 will have 'rue';
    }
}
```