CS 131
PROGRAMMING LANGUAGES
(WEEK 5)

UCLA WINTER 2019

TA: SHRUTI SHARAN

DISCUSSION SECTION: 1D

ADMINISTRATIVE INTRODUCTION

TA: Shruti Sharan

Email: shruti5596@g.ucla.edu
(Will reply by EOD)

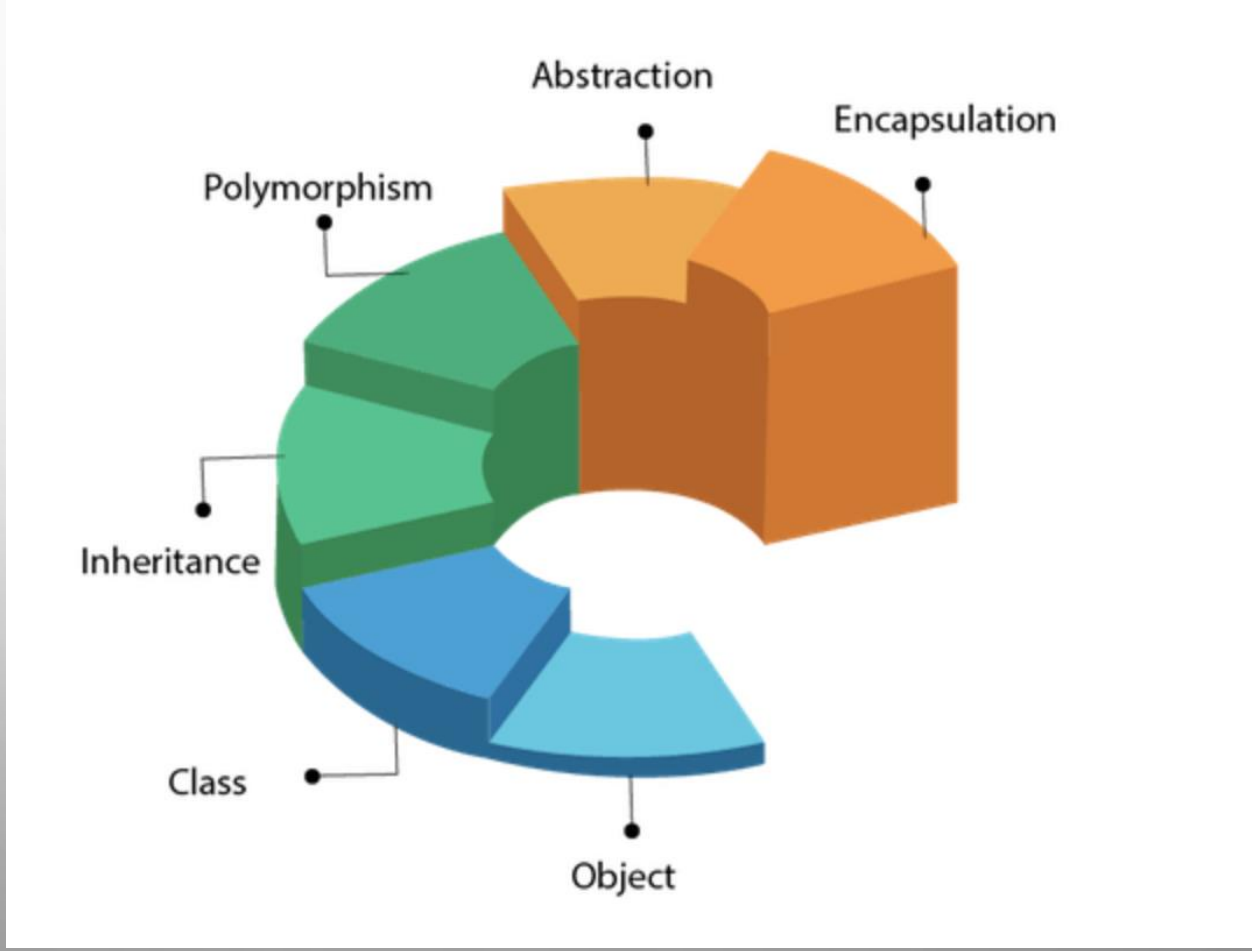
Office Hours: Mondays 1.30PM – 3.30PM
Location: Eng. VI 3rd Floor

Discussion Section: Friday 4.00-5.50PM
Location: 2214 Public Affairs

TODAY'S AGENDA

1. OBJECT ORIENTED CONCEPTS
2. INTRO-DUCTION TO JAVA
3. JAVA MEMORY MODEL
4. HOMEWORK 3

OBJECT ORIENTED PROGRAMMING



PURE OBJECT ORIENTED LANGUAGES

- Everything is an object
- A program is a set of objects telling each other what to do by sending messages
- Each object has it's own memory (made up by other objects)
- Every object has a type
- All objects of a specific type can receive the same messages

WHAT IS AN OBJECT

- An object is an encapsulation of data.
- An object has:
  - Unique reference
  - State
  - Behavior
- An object is an instance of an abstract data type
- An abstract data type is implemented as a class

OBJECTS - BENEFITS

- Modularity
  - Splitting code into objects can help keep different parts of code separated
- Information-hiding
  - Objects should only interact by using each other's public methods
  - Internal implementation hidden -> easy to change later
- Code re-use
  - Objects easy to re-use in other programs
- Pluggability and debugging ease
  - You can easily replace a buggy object with a working one if necessary

CLASSES

- Template for an object
  - Object is an instance of a class
  - E.g. We can have multiple Bicycle objects that function the same way, but can be moving at different speeds, have different specifications etc
- All objects created using the same class will have the same methods/fields

CLASS VS OBJECT

Class

- A description of a common set of objects.
- Think of this as a blueprint
- Think, examples:
  - Person
  - Board Game
  - Animal

Object

- An instance of a class.
- It is a representation of a single instance.
- Examples:
  - Paul Eggert
  - Monopoly, Exploding Kittens
  - Dog, Cat

JAVA: OBJECT ORIENTED DESIGN

- Four fundamental concepts:

Polymorphism

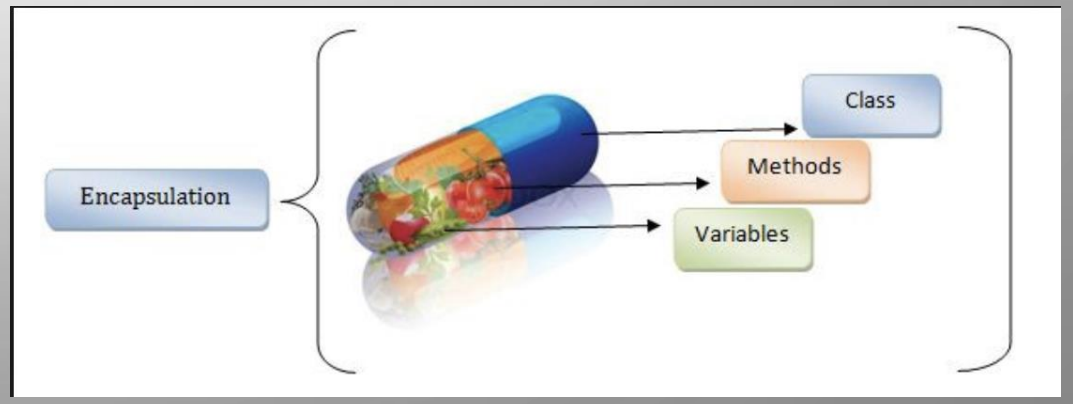
Encapsulation

Abstraction

Inheritance

ENCAPSULATION

- This is when we “encapsulate” the data, i.e hide it from the outside world.
- Data accessed with methods
- Hidden data and methods only for use within object
  - Can be changed without affecting anything outside the object
  - “Outside world” doesn’t know it exists



ABSTRACTION

- Process of hiding the implementation details from the user.
- Only the functionality will be provided to the user.

Abstract Classes

Interfaces

INTERFACE

- Interface = Forced Template
- Class-like, but no variable declarations or method bodies
- Useful for:
  - Describing methods that a class is forced to implement
  - Prescribing to a set type concept
  - Creating multiple inheritance-like hierarchies

```java
// Java program to demonstrate working of
// interface.
import java.io.*;

// A simple interface
interface in1
{
    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}

// A class that implements interface.
class testClass implements in1
{
    // Implementing the capabilities of
    // interface.
    public void display()
    {
        System.out.println("Geek");
    }

    // Driver Code
    public static void main (String[] args)
    {
        testClass t = new testClass();
        t.display();
        System.out.println(a);
    }
}
```

ABSTRACT CLASSES

- ‘abstract’ keyword is used to create an abstract class in java.
- Abstract class in java can’t be instantiated.
- We can use abstract keyword to create an abstract method, an abstract method doesn’t have body.
- The subclass of abstract class in java must implement all the abstract methods unless the subclass is also an abstract class.

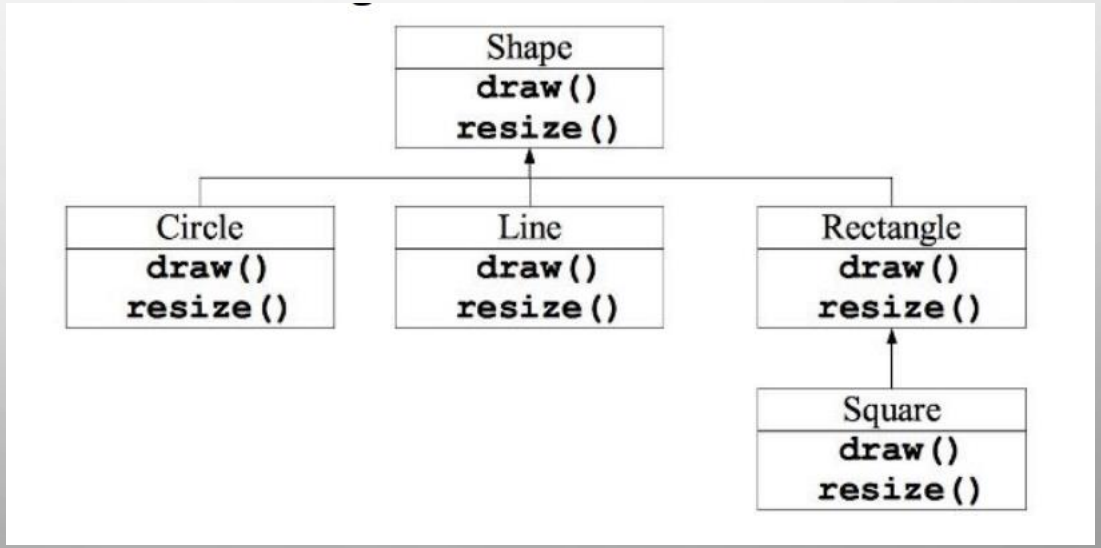
INTERFACE VS ABSTRACT CLASSES

- Methods can be declared
- No method bodies
- Constants can be declared
- Has no constructors
- Multiple inheritance possible
- Has no top interface
- Multiple parent interfaces

- Methods can be declared
- Methods can be implemented
- All types of variables can be declared
- Can have constructors
- Multiple inheritance is not possible
- Always inherits from Object
- Only one parent

INHERITANCE

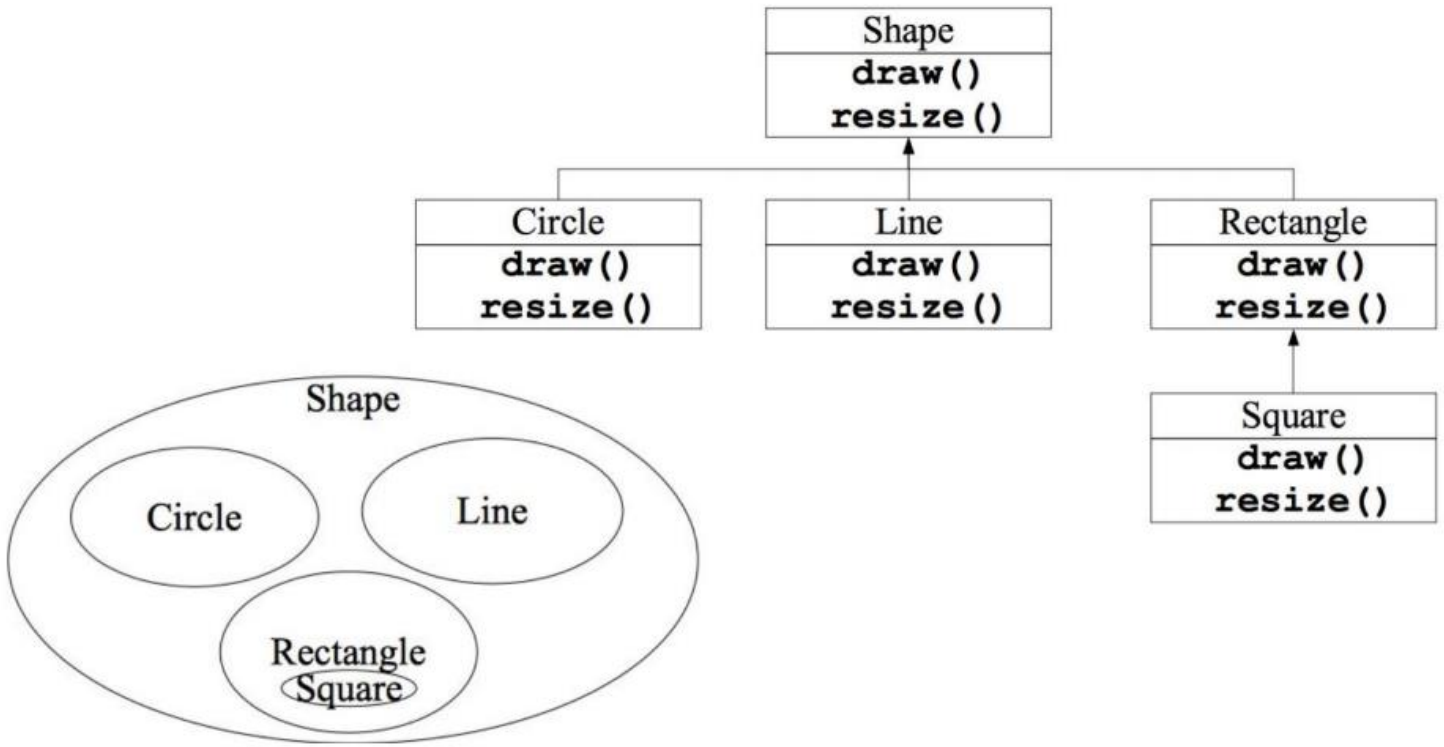
- Inheritance allows re-using and extending the functionality of a parent class
- Subclasses of a larger class



INHERITANCE

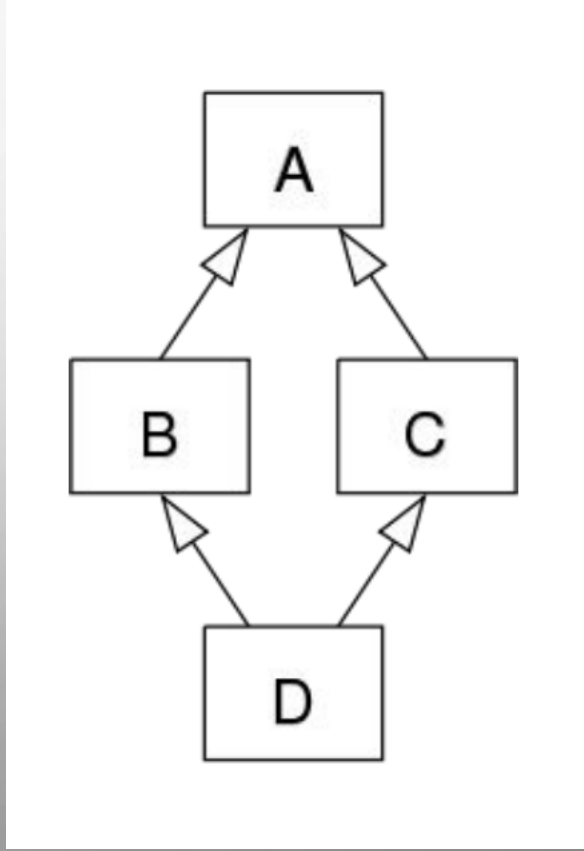
- Useful for:
  - Deriving new class from an old class
    - EX: Binary Search Tree might inherit from a generic Tree class
  - Specializing a general class
    - EX: If all dogs bark, why not have a generic Dog class that Poodle inherits from?
  - “Is-a” Relationship
    - EX: A Poodle is a Dog (but not vice-versa)

INHERITANCE - EXAMPLE



INHERITANCE

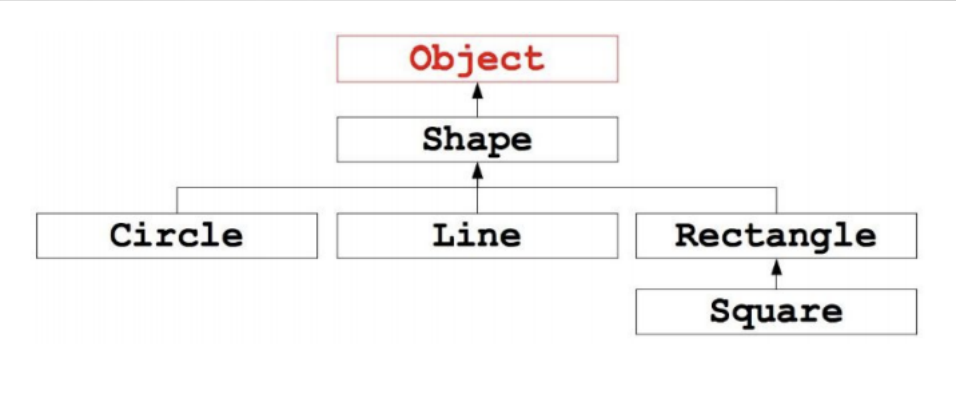
- Java has single inheritance
- Why would we want this?



Diamond Problem

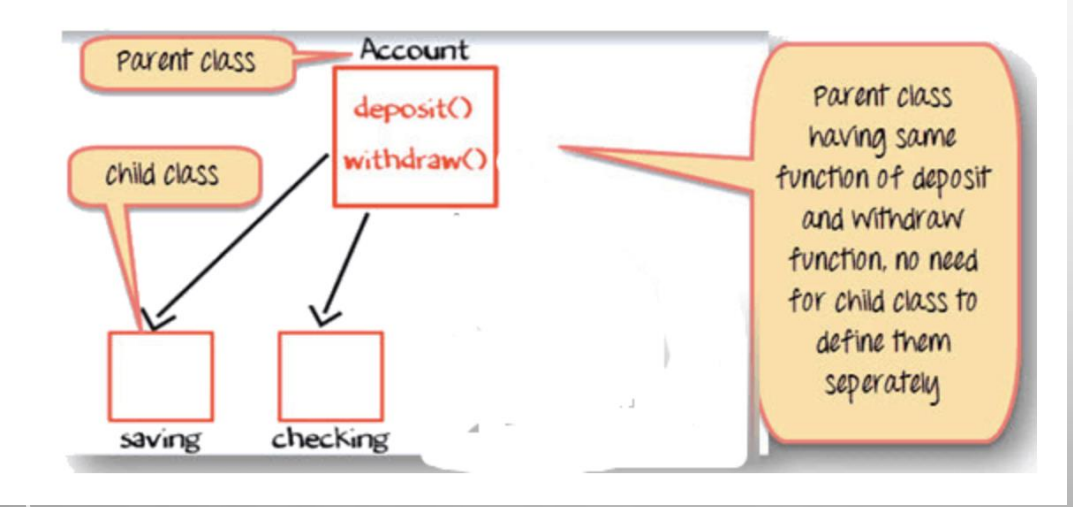
HIERARCHICAL INHERITANCE

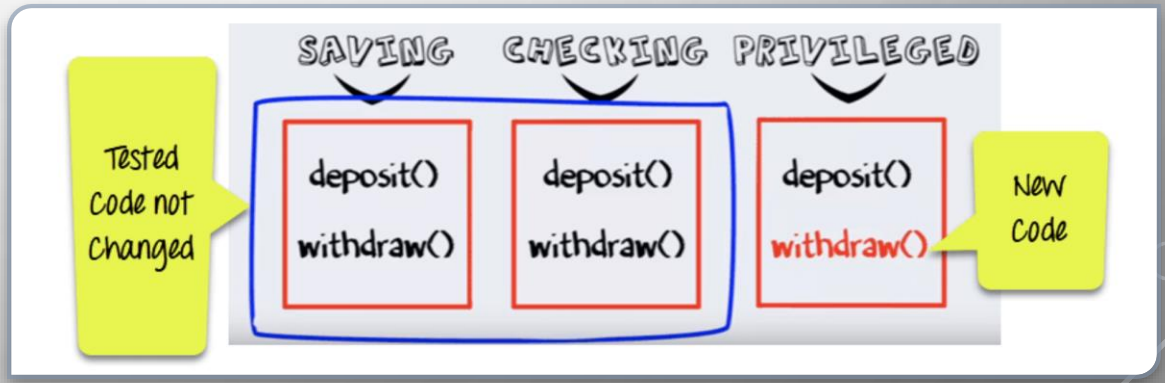
- Where does the chain of Inheritance end?
- The Object class is the superclass of all objects
- If no specified superclass, a class inherits implicitly from Object
- If superclass specified explicitly, the subclass will inherit indirectly from Object



POLYMORPHISM

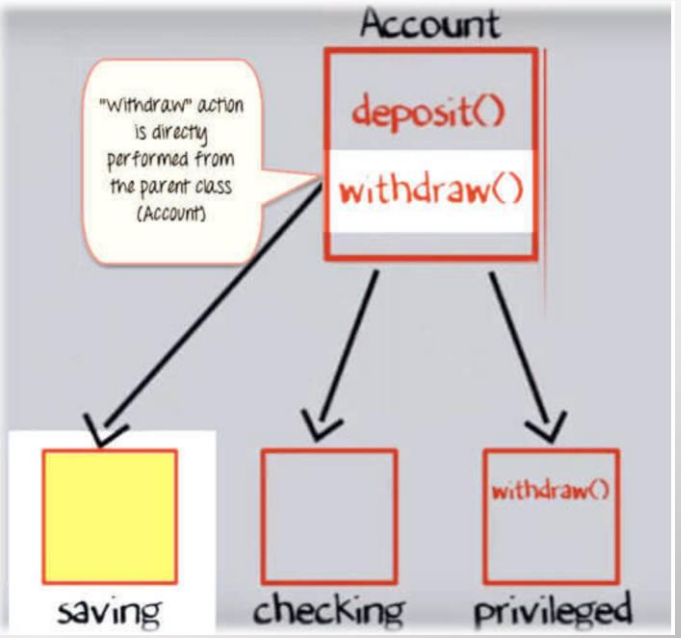
- Polymorphism is a OOPs concept where one name can have many forms

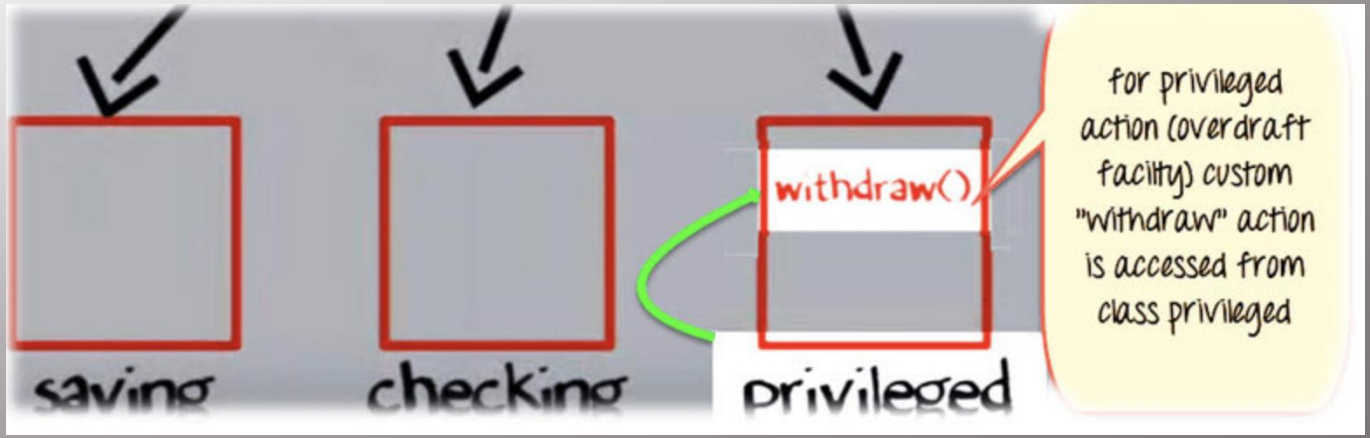




POLYMORPHISM

- Such that when the "withdrawn" method for saving account is called a method from parent account class is executed.
- But when the "Withdraw" method for the privileged account (overdraft facility) is called withdraw method defined in the privileged class is executed.





JAVA : INTRODUCTION

- We will be using Java 11 for in this class.
- Recommended that you use an IDE like Eclipse, Netbeans, IntelliJ.
  - This will give you autocomplete, debugging, syntax highlighting and other features to make your life easier
- Other option is to use a text editor with the terminal
  - Compile with: javac fileName.java
  - Run with: java fileName

JAVA VIRTUAL MACHINE (JVM)

- Runs bytecode generated by a Java compiler
- Provides separation of code and operating system / hardware
  - Write once, run everywhere
- Competing JVM implementations
  - Performance
    - Just-in-time compilation
    - Garbage collection
  - Security
  - Support for different CPU architectures
  - Support for different operating systems
- Reference implementation provided by Oracle
  - Usually the best choice

FILES

- MyClass.java = Code for MyClass
- MyClass.class = Bytecode for MyClass (Compiled from MyClass.java)
- Foo.jar = Java Archive file
  - ZIP archive
  - Could contain e.g. your whole application with all the images and other resources
  - In your homework, you are provided a jar file containing the necessary code files

HELLO, WORLD!

```
class HelloWorld
{
    public static void main(String args[])
    {
    System.out.println("Hello World");
    }
}
```

HOW TO RUN THE CODE

- Insert the code for HelloWorld class inside a HelloWorld.java file
- Execute javac HelloWorld.java
  - This generates a HelloWorld.class file, containing the bytecode
- Run your code with java HelloWorld
  - Note, the parameter is your class name, not the file name

```
[Shruti@Documents $  javac HelloWorld.java
[Shruti@Documents $  java HelloWorld
Hello World
```

ACCESS MODIFIERS

- Controlling who can access object's methods/fields

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

PARAMETER PASSING

- In java, parameter is passed by value.
- The value of the actual parameter is copied into the formal parameter.
- Values are always pointers except for primitives.
- This is basically saying that value of actual parameter cannot be changed by a function call.

```
int add(int x, int y) {
        x +=y;
        return x;
}

void foo() {
        int x = 4;
        int y = 2;
        int z = add(x,y);
}
```

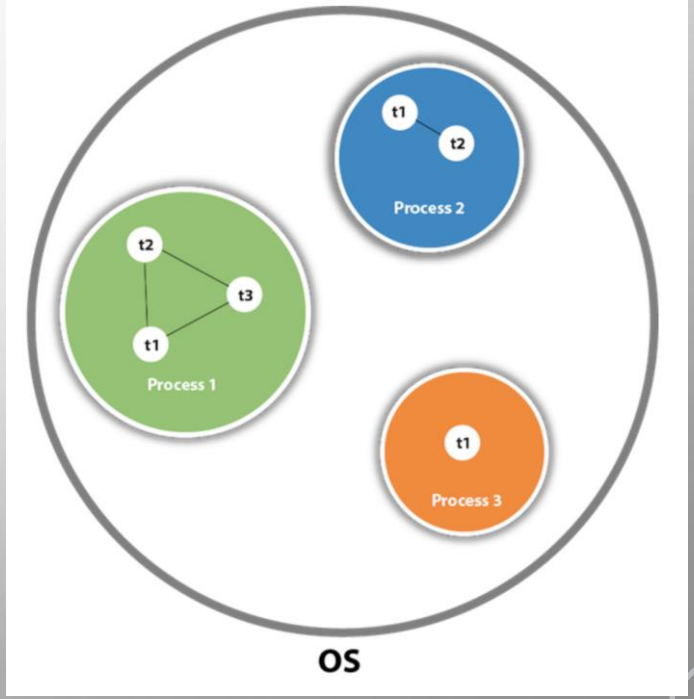
```
class Integer {
    int i;
    Integer(int val) { this.i = val; }
}

Integer new_plus(Integer a, Integer b) {
    a = new Integer(a.i + b.i);
    return a;
}

Integer plus(Integer a, Integer b) {
    a.i += b.i;
    return a;
}
```

```
void foo() {
    Integer x = new Integer(4);
    Integer y = new Integer(3);
    Integer z = new_plus(x,y);
    Integer s = plus(x,y)
}
```

- After calling new_plus x still points to original and is unchanged. z points to a new object with value 7 in it.
- After calling plus both x and s point to the same object that pointed to x. Now this object holds value 7 instead of 4

THREADING

- A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads.
- It uses a shared memory area.
- A thread is executed inside the process.
- There is context-switching between the threads.
- There can be multiple processes inside the OS, and one process can have multiple threads.



THREAD CLASS VS RUNNABLE INTERFACE

- Threads can be created by using two mechanisms :
  1. Extending the **Thread** class
  2. Implementing the **Runnable** Interface

What's the difference?

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

PROBLEMS WITH THREADING

- Consider a case where two threads want to increase a counter simultaneously:

Thread 1:

Counter+=1;

Thread 2:

Counter+=2;

RACE CONDITIONS

- A race condition occurs when two or more threads can access shared data and they try to change it at the same time.
- Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data.
- Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

PROBLEMS WITH THREADING

- In how many ways can these commands be ordered?

Thread 1:

```
tmp1 = counter + 1;
counter = tmp;
```

Thread 2:

```
tmp2 = counter + 2;
counter = tmp;
```

tmp1 = cnt + 1;	
cnt = tmp1;	
	tmp2 = cnt + 2;
	cnt = tmp2;

cnt <- cnt + 3

	tmp2 = cnt + 2;
	cnt = tmp2;
tmp1 = cnt + 1;	
cnt = tmp1;	

cnt <- cnt + 3

tmp1 = cnt + 1;	
	tmp2 = cnt + 2;
cnt = tmp1;	
	cnt = tmp2;

cnt <- cnt + 2

tmp1 = cnt + 1;	
	tmp2 = cnt + 2;
	cnt = tmp2;
cnt = tmp1;	

cnt <- cnt + 1

	tmp2 = cnt + 2;
tmp1 = cnt + 1;	
	cnt = tmp2;
cnt = tmp1;	

cnt <- cnt + 1

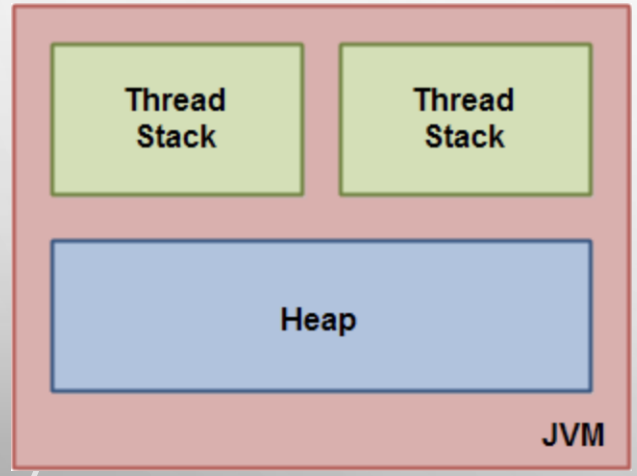
	tmp2 = cnt + 2;
tmp1 = cnt + 1;	
cnt = tmp1;	
	cnt = tmp2;

cnt <- cnt + 2

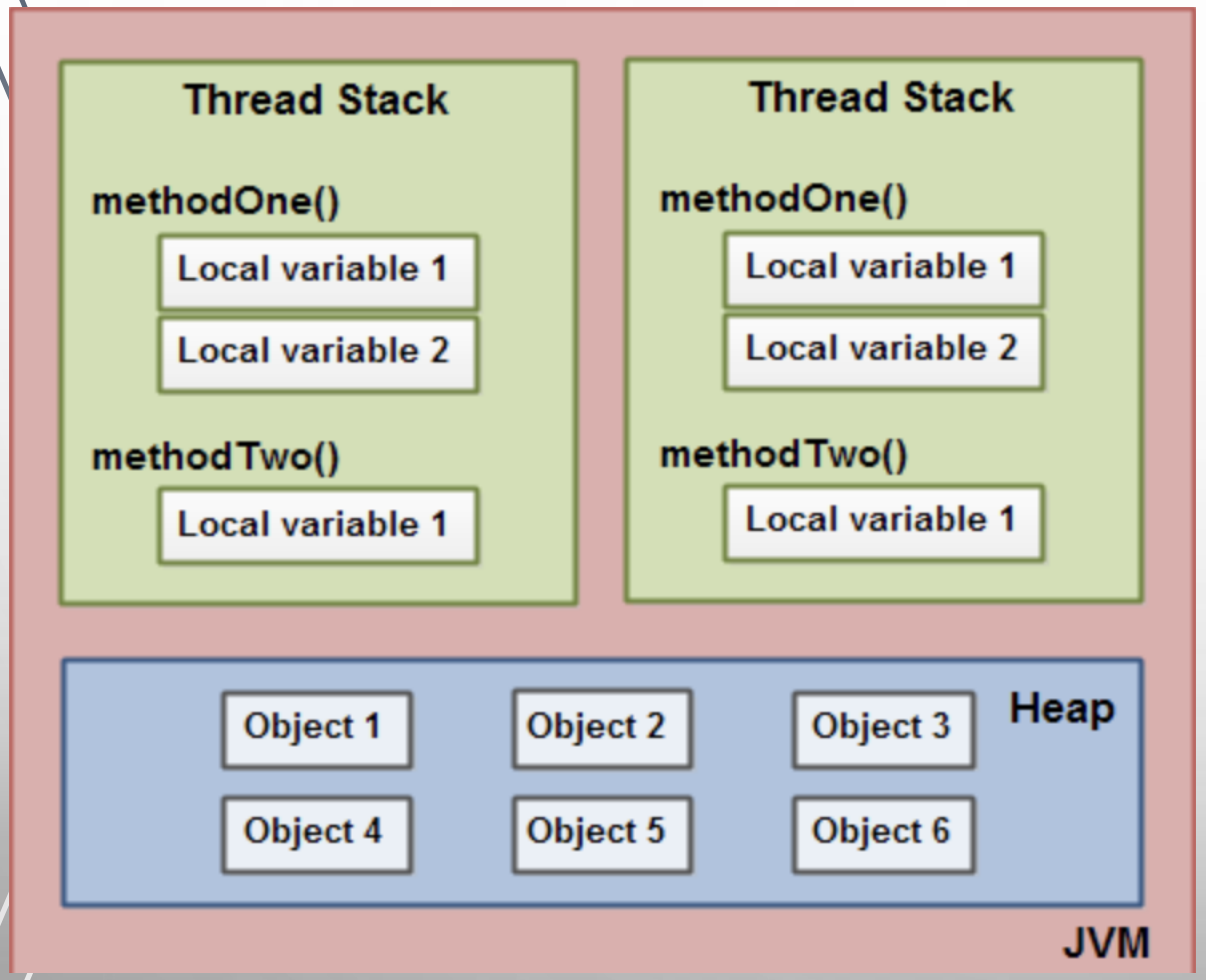
JAVA MEMORY MODEL

- The Java memory model specifies how and when different threads can see values written to shared variables by other threads and how to synchronize access to shared variables when necessary.
- JMM describes how Java is guaranteed to behave in different situations
- Uses happens-before to describe what the order of statements will be

STACK AND HEAP



- Each thread running in the Java virtual machine has its own thread stack.
- Contains information about methods the thread has called to reach the current point of execution.
- Contains all local variables for each method being executed.
- A thread can only access it's own thread stack
- Even if two threads are executing the exact same code, the two threads will create their own local variables.
- The heap contains all objects created in your Java application, regardless of what thread created the object.



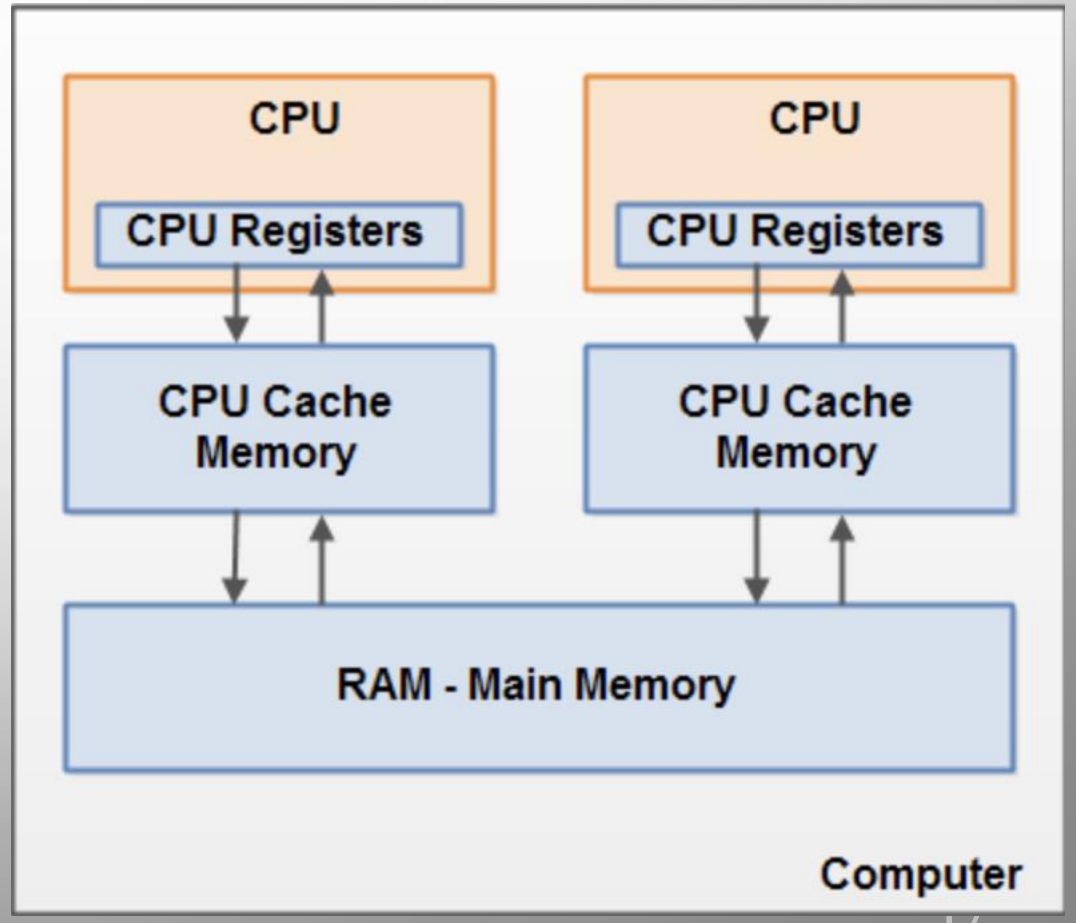
- A local variable may be of a primitive type, in which case it is totally kept on the thread stack.
- A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on the thread stack, but the object itself if stored on the heap.
- Static class variables are also stored on the heap along with the class definition.

EXAMPLE

```
Set s = new ListSet();

// s is on the stack   the new object, call it o, is on the heap
//  s is a pointer to the new object

s.add("hi");

// dereferences s and updates the object o
//  to contain the "hi" value

Set s2 = s;

// s2 now points to o
// s and s2 are *aliases*

s2.remove("hi");

// now o no longer contains "hi"
```

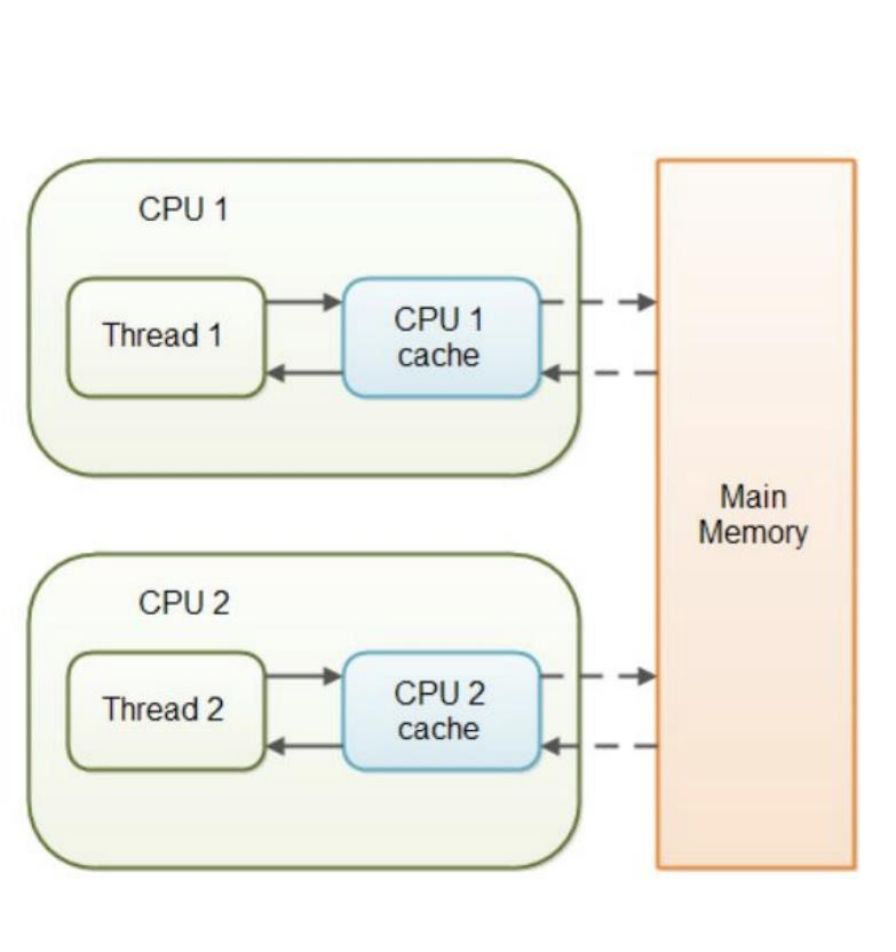
HARDWARE MEMORY ARCHITECTURE

- Each CPU is capable of running one thread at any given time.
- The CPU can perform operations much faster on these registers than it can perform on
- When a CPU needs to access main memory it will read part of main memory into its CPU cache. It may even read part of the cache into its internal registers and then perform operations on it.
- When the CPU needs to write the result back to main memory it will flush the value from its internal register to the cache memory, and at some point flush the value back to main memory.



BRIDGING THE GAP BETWEEN THE JAVA MEMORY MODEL AND THE HARDWARE MEMORY ARCHITECTURE

- The Java memory model and the hardware memory architecture are different.
- The hardware memory architecture does not distinguish between thread stacks and heap.
- The two main problems are:
  - Visibility of thread updates (writes) to shared variables.
  - Race conditions when reading, checking and writing shared variables.



synchronized KEYWORD

- Block of code that is marked as synchronized in Java tells the JVM: "only let one thread in here at a time”
- Only one thread can hold a monitor at a time, others will wait.

```
public class SynchronizedCounter {
  private int c = 0;
  public synchronized void increment() { c++; }
  public synchronized void decrement() { c--; }
  public synchronized int value() { return c; }
}
```

Let **count** be an instance of SynchronizedCounter.
If thread1 is using count.increment(), thread2 can neither count.increment(), count.decrement() nor getCount().

Volatile KEYWORD

- volatile is used to indicate that a variable's value will be modified by different threads.
- The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory"
- Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.
- A primitive variable may be declared volatile (whereas you can't synchronize on a primitive with synchronized)

- Unlike a synchronized block we will never hold on to any lock -- **no blocking.**
- Because accessing a volatile variable **never holds a lock,** it is not suitable for cases where we want to **read-update-write** as an atomic operation (unless we're prepared to "miss an update")
- A volatile variable that is an object reference may be null (because you're effectively synchronizing on the reference, not the actual object)

EXAMPLE

- Using the **volatile** keyword

note that "static" means the variable is associated with the class, not with the instance.

```
public class VolatileExample {
    private static volatile int MY_INT = 0;
    static class ChangeMaker extends Thread {
        public void run() {
                // make some change to MY_INT
        }
    static class ChangeListener extends Thread {
        public void run() {
                // looks for changes to MY_INT (by ChangeMaker)
                // If we don't make MY_INT volatile, we cannot
                // guarentee change is detected
        }
}
```

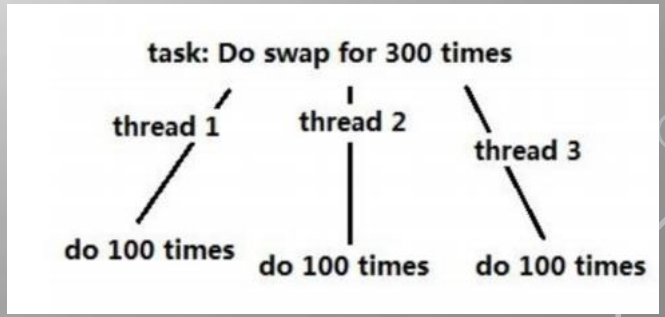
HOMEWORK #3

- Thread safety vs. Performance
- You'll be comparing different approaches to do synchronization between threads - Some approaches are faster than others, some approaches have 100% safety guarantee, some do not.

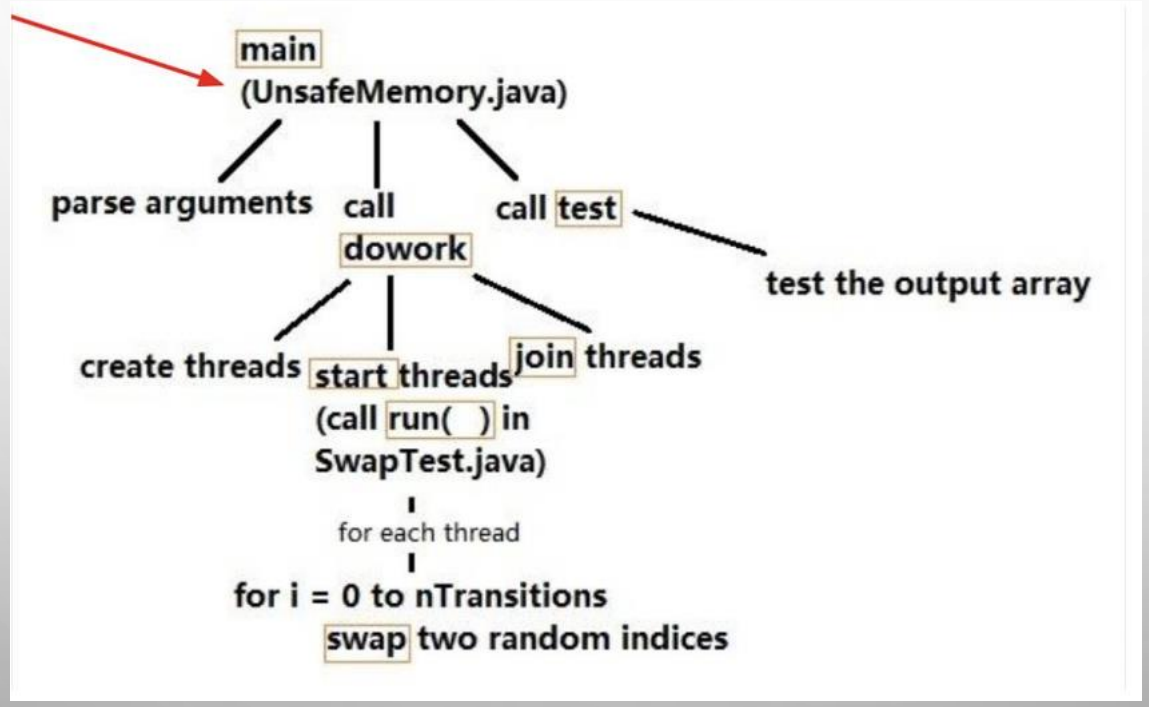
BACKGROUND

0	1	2	3	4	5	6	7
27	2	101	84	27	19	30	3

- We have an array containing integer values between 0-maxval: -
- We want to perform N swap(i,j) operations, which decrease ith value by 1 and increase jth value by 1
- To make this faster, we want to split the operations into multiple threads
  - E.g. with 3 threads, each thread performs N/3 swap operations
- Sum of all the integers should remain constant.
  - But if it doesn't - we have a syncronization problem
- Each value should remain between the given bounds (0-maxval)



EXAMPLE CODE - OVERVIEW



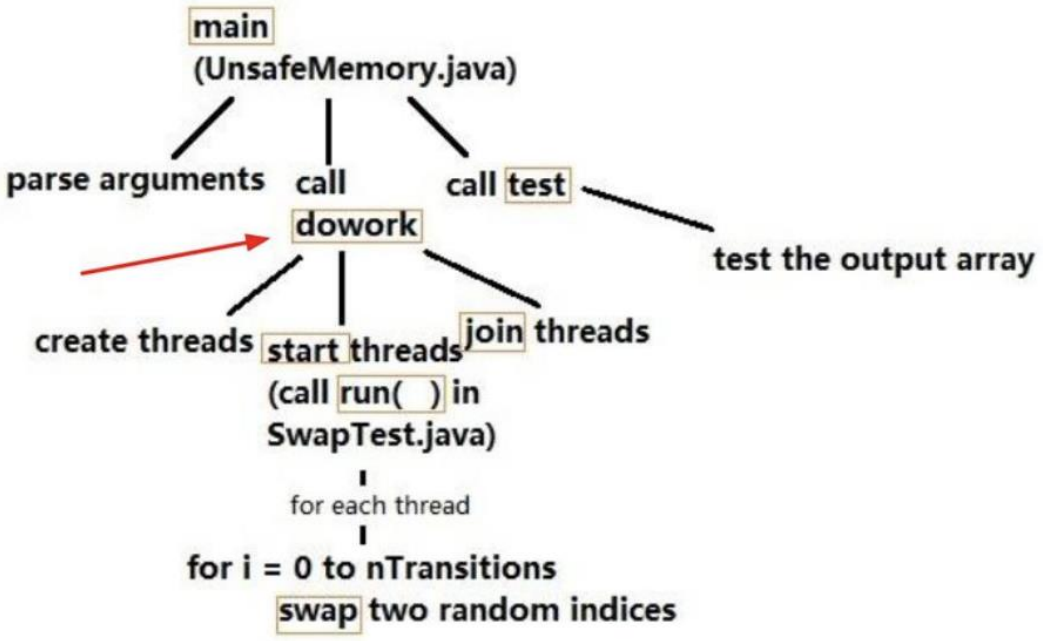
Parse arguments

Initialize the state object

Call dowork()

Call test()

```
class UnsafeMemory {
    public static void main(String args[]) {
        if (args.length < 3)
            usage(null);
        try {
            int nThreads = parseInt (args[1], 1, Integer.MAX_VALUE);
            int nTransitions = parseInt (args[2], 0, Integer.MAX_VALUE);
            byte maxval = (byte) parseInt (args[3], 0, 127);
            byte[] value = new byte[args.length - 4];
            for (int i = 4; i < args.length; i++)
                value[i - 4] = (byte) parseInt (args[i], 0, maxval);
            byte[] stateArg = value.clone();
            State s;
            if (args[0].equals("Null"))
                s = new NullState(stateArg, maxval);
            else if (args[0].equals("Synchronized"))
                s = new SynchronizedState(stateArg, maxval);
            else
                throw new Exception(args[0]);
            dowork(nThreads, nTransitions, s);
            test(value, s.current(), maxval);
            System.exit (0);
        } catch (Exception e) {
            usage(e);
```


main
(UnsafeMemory.java)
parse arguments
call
dowork
call test
test the output array
create threads
start threads
(call run( ) in
SwapTest.java)
join threads
for each thread
for i = 0 to nTransitions
swap two random indices

Create Threads

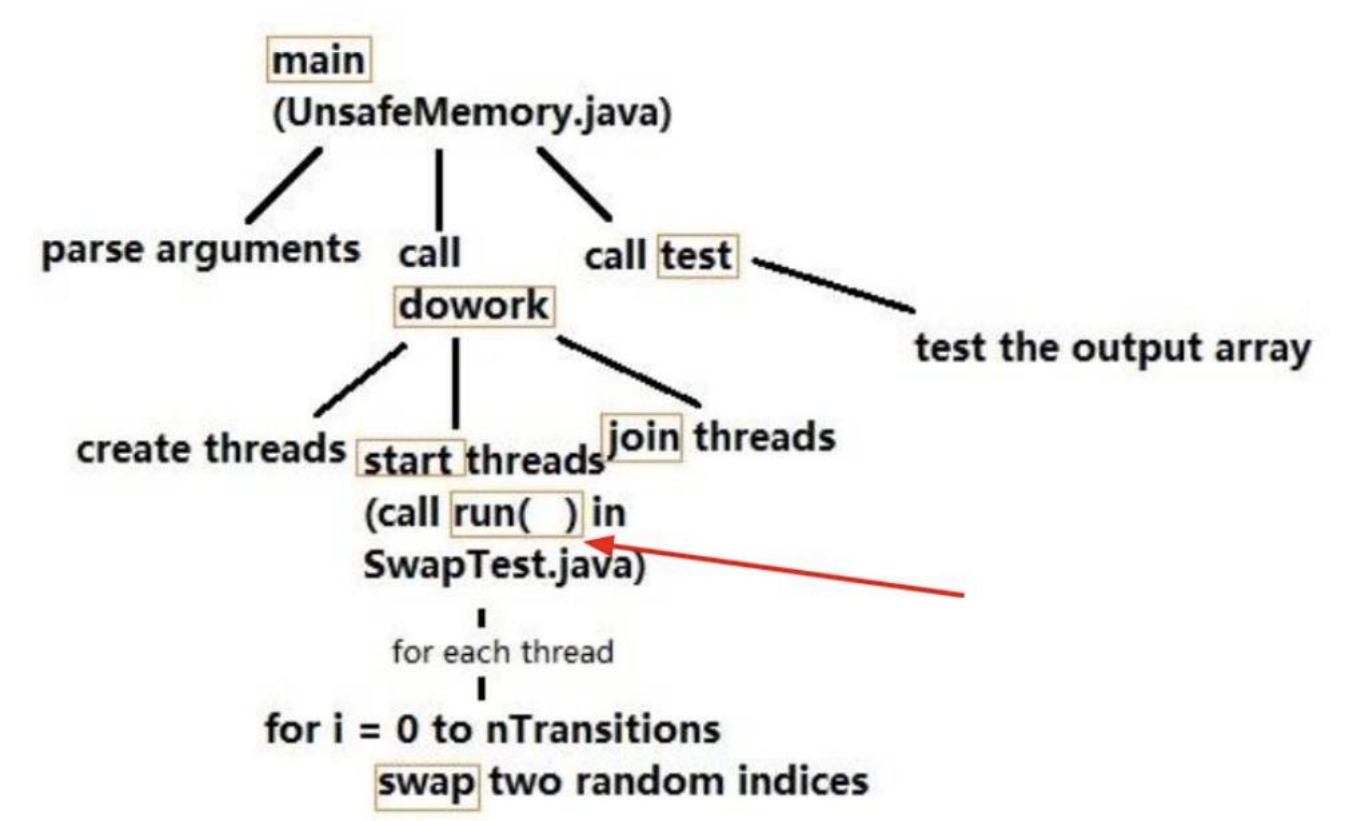
Start Threads

Join Threads

```
private static void dowork(int nThreads, int nTransitions, State s)
  throws InterruptedException {
    Thread[] t = new Thread[nThreads];
    for (int i = 0; i < nThreads; i++) {
        int threadTransitions =
            (nTransitions / nThreads
             + (i < nTransitions % nThreads ? 1 : 0));
        t[i] = new Thread (new SwapTest (threadTransitions, s));
    }
    long start = System.nanoTime();
    for (int i = 0; i < nThreads; i++)
        t[i].start ();
    for (int i = 0; i < nThreads; i++)
        t[i].join ();
    long end = System.nanoTime();
    double elapsed_ns = end - start;
    System.out.format("Threads average %g ns/transition\n",
                      elapsed_ns * nThreads / nTransitions);
}
```

Amount of work done by each thread

Wait for threads to finish (keeping track of time).

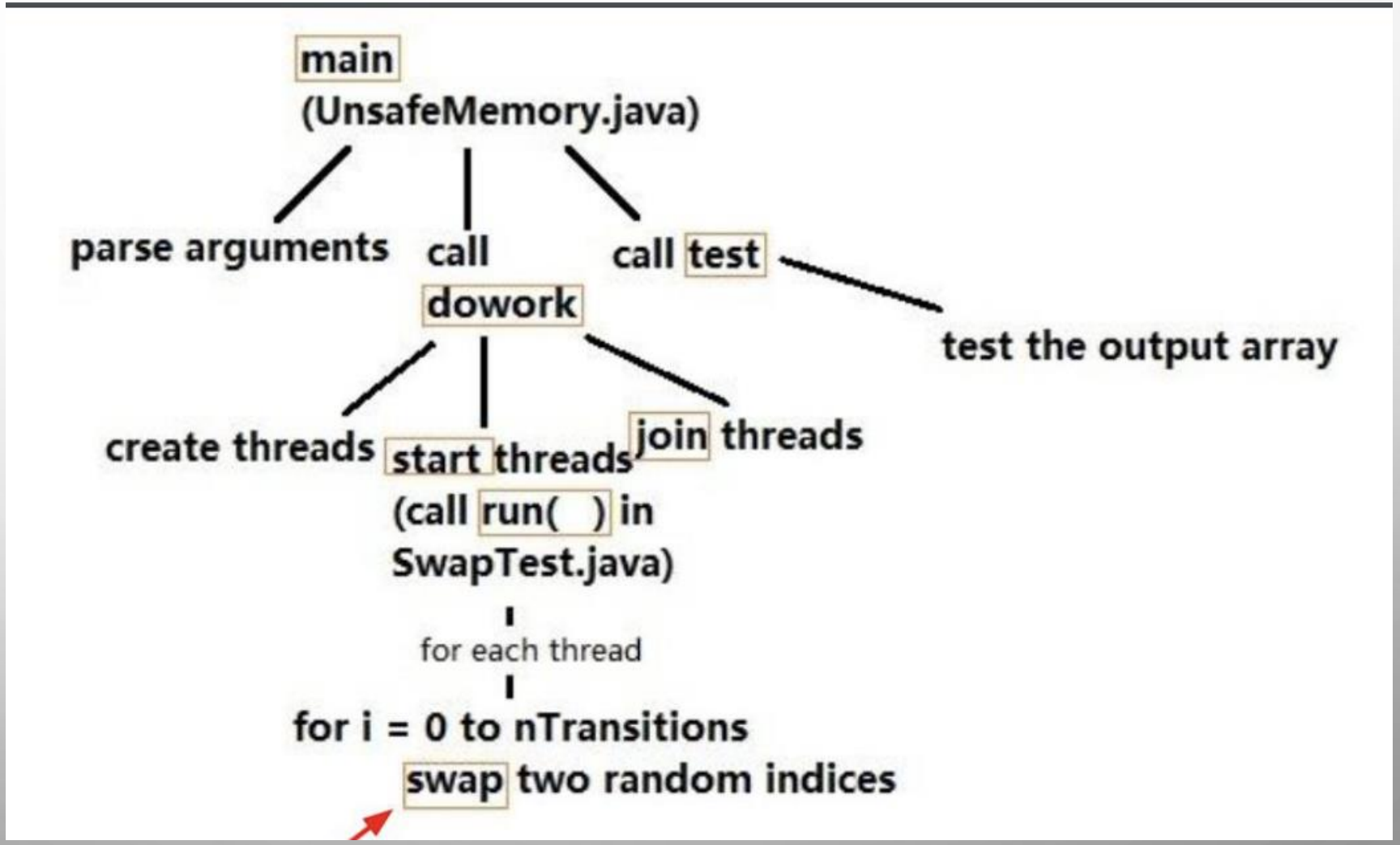

main
(UnsafeMemory.java)
parse arguments
call
dowork
call test
test the output array
create threads
start threads
(call run( ) in
SwapTest.java)
join threads
for each thread
for i = 0 to nTransitions
swap two random indices

Contains test code for one thread:

runs state.Swap(a,b) with random values a and b as many times as specified

```
import java.util.concurrent.ThreadLocalRandom;

class SwapTest implements Runnable {
    private int nTransitions;
    private State state;

    SwapTest(int n, State s) {
        nTransitions = n;
        state = s;
    }

    public void run() {
        int n = state.size();
        if (n != 0)
            for (int i = 0; i < nTransitions; ) {
                int a = ThreadLocalRandom.current().nextInt(0, n);
                int b = ThreadLocalRandom.current().nextInt(0, n - 1);
                if (a == b)
                    b = n - 1;
                if (state.swap(a, b))
                    i++;
            }
    }
}
```

- Runnable is an interface.
- To implement it, our class must have a method called **run()**
- Runnable interface defines that a Thread object can run this code


main
(UnsafeMemory.java)
parse arguments
call
dowork
call test
test the output array
create threads
start threads
(call run( ) in
SwapTest.java)
join threads
for each thread
for i = 0 to nTransitions
swap two random indices

DATA STRUCTURE – State.java

```
interface State {
    int size();
    byte[] current();
    boolean swap(int i, int j);
}
```

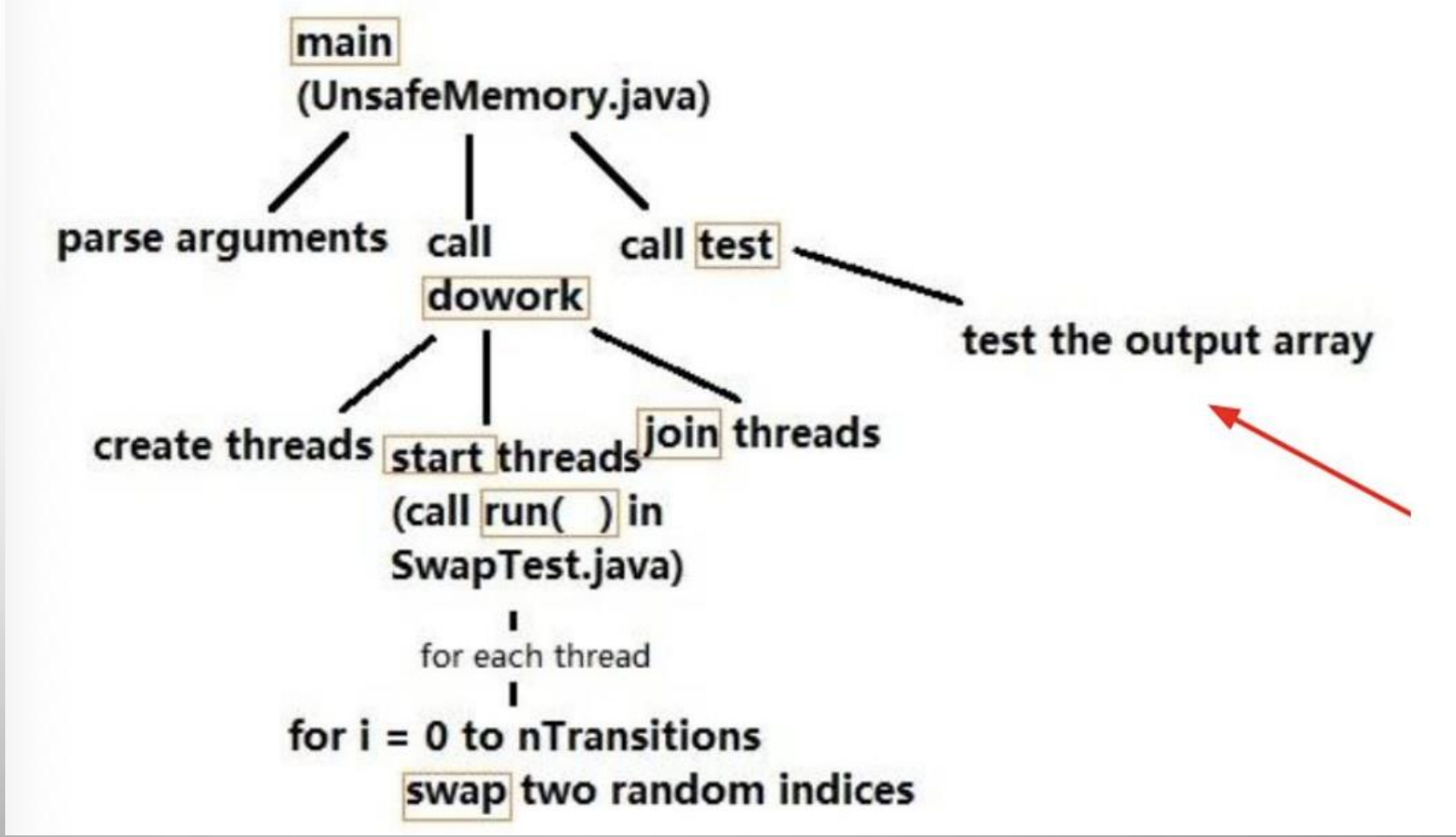
Returns the array

Does the swap operation on indices I and j

```
class NullState implements State {
    private byte[] value;
    NullState(byte[] v, byte maxval) { value = v; }
    public int size() { return value.length; }
    public byte[] current() { return value; }
    public boolean swap(int i, int j) { return true; }
}
```

- This is a dummy implementation
- useful for deducing the overhead of the testing framework

```java
class SynchronizedState implements State {
    private byte[] value;
    private byte maxval;

    SynchronizedState(byte[] v) { value = v; maxval = 127; }

    SynchronizedState(byte[] v, byte m) { value = v; maxval = m; }

    public int size() { return value.length; }

    public byte[] current() { return value; }

    public synchronized boolean swap(int i, int j) {
        if (value[i] <= 0 || value[j] >= maxval) {
            return false;
        }
        value[i]--;
        value[j]++;
        return true;
    }
}
```


main
(UnsafeMemory.java)
parse arguments
call
dowork
call test
test the output array
create threads
start threads
(call run( ) in
SwapTest.java)
join threads
for each thread
for i = 0 to nTransitions
swap two random indices

Verify that the state is consistent

Check if input length is equal to output length

Check every element in output list does not exceed maxval or go below 0

Check sum of elements in output is the same as sum of elements in input.

```
private static void test(byte[] input, byte[] output, byte maxval) {
    if (input.length != output.length)
        error("length mismatch", input.length, output.length);
    long isum = 0;
    long osum = 0;
    for (int i = 0; i < input.length; i++)
        {
            isum += input[i];
            osum += output[i];
            if (output[i] < 0)
                error("negative output", output[i], 0);
            if (output[i] > maxval)
                error("output too large", output[i], maxval);
        }
    if (isum != osum)
        error("sum mismatch", isum, osum);
}
```

TASK #1

- Implement an unsynchronized state class
  - Similar to SynchronizedState.java, except without the synchronized keyword
- This solution does not guarantee any thread safety!

TASK #2

- Implement GetNSet, which is a compromise between synchronized and unsynchronized state classes
- You should use AtomicIntegerArray class, which provides atomic updates

TASK #3

- Design and implement a thread-safe class *BetterSafe*, which is faster than the synchronized class
- You should read Using JDK 9 Memory Order Modes
- Also, check packages:
  - java.util.concurrency
  - java.util.concurrent.atomic
  - java.util.concurrent.locks
  - java.lang.invoke.VarHandle

TASK #4

- Integrate all the state classes into one program UnsafeMemory
  - Basically, check how *Synchronized* class is used in *UnsafeMemory* and add your classes in the same way.

TASK #5 & #6

- Measure and characterize the performance and reliability of each class
- Compare these measurements

REPORT

- Write a 2-3 page report discussing:
  - Pros/cons of the four packages that you were given for BetterSafe implementation
  - Why your solution is faster than Synchronized and why it is still 100% reliable
  - Discuss any problems you had to overcome to do your measurements properly
  - Explain why your class is free of data races, or if it isn't (due to a bug), show how to reproduce the problem (i.e. how to run the program in a way that it is likely to fail)

REFERENCES

- Useful Reading:
- https://en.wikibooks.org/wiki/C%2B%2B_Programming/Programming_Languages/Comparisons/Java
- https://www.cprogramming.com/tutorial/java/syntax-differences-java-c++.html
- https://www.oracle.com/technetwork/java/javase/11-relnote-issues-5012449.html
- https://www.cs.umd.edu/~pugh/java/memoryModel/
- https://beginnersbook.com/2013/04/java-static-class-block-methods-variables/
- https://www.javaworld.com/article/2077424/learn-java/does-java-pass-by-reference-or-pass-by-value.html
- https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/locks/Lock.html
- http://tutorials.jenkov.com/java-concurrency/volatile.html
- https://www.ibm.com/developerworks/library/j-5things15/index.html
- http://www.baeldung.com/java-variable-handles