Core Java

Agenda

- C-DAC Project discussion
- Multi-tasking concepts
- Java processes
- Java Multi-threading

C-DAC Project discussion

- Not important:
  - Project title, idea.
  - Project technology.
- Important:
  - Your project contribution
  - Your project explanation/code
  - Project deployment
- Steps
  - Decide project group -- Max 4
    - Must be in student group (KDx).
  - Decide project title and short description.
  - Requirement analysis
    - Study existing websites.
    - Decide the user roles e.g. e-commerce appln: Buyer, Seller, Delivery, Support.
    - Decide functionalities in your application for each role.
    - Output: Use-case diagram.
  - UI design
    - Web pages design for each operation in use case.
      - Detailed: Textboxes, Dropdown, Checkboxes, Radiobuttons, Buttons, ...

        - Flow
        - Output: UI design diagrams
    - Database design
        - Normalize each transaction (do major txs first)
        - Output: ER diagram
        - Add proper test data to get started
    - API/Services design
        - Number of REST apis
        - For each REST api
            - URL
            - Request method: GET/POST/PUT/DELETE
            - Request body: JSON
            - Response body: JSON
    - Implementation
        - BackEnd and FrontEnd implementation
    - Deployment:
        - AWS cloud
        - Docker
- Technologies
    - Database: MySQL or Mongo
    - FrontEnd: React
    - BackEnd: Java Spring, .NET, Express JS
- Evaluations
    - Continuous Evaluation
- Tools
    - draw.io
    - github.com
    - Advanced: Jira

Platform Independence

- Java is architecture neutral i.e. can work on various CPU architectures like x86, ARM, SPARC, PPC, etc (if JVM is available on those architectures).

- Java is NOT fully platform independent. It can work on various platforms like Windows, Linux, Mac, UNIX, etc (if JVM is available on those platforms).
- Few features of Java remains platform dependent.
    - Multi-threading (Scheduling, Priority)
    - File IO (Performance, File types, Paths)
    - AWT GUI (Look & Feel)
    - Networking (Socket connection)

Multi-tasking concepts

- Multi-tasking: Executing multiple tasks/operations concurrently.
    - Types: Process-based Multi-tasking and Thread-based Multi-tasking
- Process-based Multi-tasking
    - Executing multiple independent processes concurrently.
    - Example: Running notepad and paintbrush concurrently.
    - Example: Chrome browser create new process for each tab.
    - Process = Program in execution.
    - Process = Text + Data + Rodata + Stack + Heap + PCB
- Thread-based Multi-tasking
    - Executing multiple tasks concurrently within a process.
    - Example: YouTube player: Download and play media concurrently.
    - Example: Firefox browser create new thread for each tab.
    - Thread = Lightweight process = Needs lesser resources.
    - Thread = Shared (Text + Data + Rodata + Heap) + Stack + TCB

Program

- Program is set of instructions given to the computer.
- Executable file is a program.
- Executable file contains text, data, rodata, symbol table, exe header.

Process

- Process is program in execution.

- Program (executable file) is loaded in RAM (from disk) for execution. Also OS keep information required for execution of the program in a struct called PCB (Process Control Block).
- Process contains text, data, rodata, stack, and heap section.

Thread

- Threads are used to do multiple tasks concurrently within a single process.
- Thread is a lightweight process.
- When a new thread is created, a new TCB is created along with a new stack. Remaining sections are shared with parent process.

Process vs Thread

- Process is a container that holds resources required for execution and thread is unit of execution/scheduling.
- Each process have one thread created by default -- called as main thread.

Process creation (Java)

- In Java, process can be created using Runtime object.
- Runtime object holds information of current runtime environment that includes number of processors, JVM memory usage, etc.
- Current runtime can be accessed using static getRuntime() method.

```
Runtime rt = Runtime.getRuntime();
```

- The process is created using exec() method, which returns the Process object. This object represents the OS process and its waitFor() method wait for the process termination (and returns exit status).

```
String[] args = { "/path/of/executable", "cmd-line arg1", ... };
Process p = Runtime.exec(args);
int exitStatus = p.waitFor();
```

Multi-threading (Java)

- Java applications are always multi-threaded.
- When any java application is executed, JVM creates (at least) two threads.
  - main thread -- executes the application main()
  - GC thread -- does garbage collection (release unreferenced objects)
- Programmer may create additional threads, if required.

Thread creation

- To create a thread
  - step 1: Implement a thread function (task to be done by the thread)
  - step 2: Create a thread (with above function)
- Method 1: extends Thread

```
class MyThread extends Thread {
    @Override
    public void run() {
        // task to be done by the thread
    }
}
```

```
MyThread th = new MyThread();
th.start();
```

- Method 2: implements Runnable

```
class MyRunnable implements Runnable {
    @Override
```

```
    public void run() {
        // task to be done by the thread
    }
}
```

```
MyRunnable runnable = new MyRunnable();
Thread th = new Thread(runnable);
th.start();
```

- Java doesn't support multiple inheritance. If your class is already inherited from a super class, you cannot extend it from Thread class. Prefer Runnable in this case; otherwise you may choose any method.

```
// In Java GUI application is inherited from Frame class.
// to create run() in the same class, you must use Runnable
class MyGuiApplication extends Frame implements Runnable {
    // ...
    public void run() {
        // ...
    }
    // ...
}
```

start() vs run()

- run():
  - Programmer implemented code to be executed by the thread.
- start():
  - Pre-defined method in Thread class.

  - When called, the thread object is submitted to the (JVM/OS) scheduler. Then scheduler select the thread for execution and thread executes its run() method.

Thread methods

- static Thread currentThread()
  - Returns a reference to the currently executing thread object.
- static void sleep(long millis)
  - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- static void yield()
  - A hint to the scheduler that the current thread is willing to yield its current use of a processor.
- Thread.State getState()
  - Returns the state of this thread.
  - State can be NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
- void run()
  - If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called. If thread class extends from Thread class, this method should be overridden. The default implementation is empty.
- void start()
  - Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
- void join()
  - Waits for this thread to die/complete.
- boolean isAlive()

- Tests if this thread is alive.

- void setDaemon(boolean daemon);

  - Marks this thread as either a daemon thread (true) or a user thread (false).

- boolean isDaemon()

  - Tests if this thread is a daemon thread.

- long getId()

  - Returns the identifier of this Thread.

- void setName(String name)

  - Changes the name of this thread to be equal to the argument name.

- String getName()

  - Returns this thread's name.

- void setPriority(int newPriority)

  - Changes the priority of this thread.
  - In Java thread priority can be 1 to 10.
  - May use predefined constants MIN_PRIORITY(1), NORM_PRIORITY(5), MAX_PRIORITY(10).

- int getPriority()

  - Returns this thread's priority.

- ThreadGroup getThreadGroup()

  - Returns the thread group to which this thread belongs.

- void interrupt()

  - Interrupts this thread -- will raise InterruptedException in the thread.

- boolean isInterrupted()

  - Tests whether this thread has been interrupted.

Daemon threads

- By default all threads are non-daemon threads (including main thread).
- We can make a thread as daemon by calling its setDaemon(true) method -- before starting the thread.
- Daemon threads are also called as background threads and they support/help the non-daemon threads.
- When all non-daemon threads are terminated, the Daemon threads get automatically terminated.

Thread life cycle

- Thread.State state = th.getState();
- NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
  - NEW: New thread object created (not yet started its execution).
  - RUNNABLE: Thread is running on CPU or ready for execution. Scheduler picks ready thread and dispatch it on CPU.
  - BLOCKED: Thread is waiting for lock to be released. Thread blocks due to synchronized block/method.
  - WAITING: Thread is waiting for the notification. Waiting thread release the acquired lock.
  - TIMED_WAITING: Thread is waiting for the notification or timeout duration. Waiting thread release the acquired lock.
  - TERMINATED: Thread terminates when run() method is completed, stopped explicitly using stop(), or an exception is raised while executing run().

Synchronization

- When multiple threads try to access same resource at the same time, it is called as Race condition.
- Example: Same bank account undergo deposit() and withdraw() operations simultaneously.
- It may yield in unexpected/undesired results.
- This problem can be solved by Synchronization.
- The synchronized keyword in Java provides thread-safe access.
- Java synchronization internally use the Monitor object associated with any object. It provides lock/unlock mechanism.
- "synchronized" can be used for block or method.

- It acquires lock on associated object at the start of block/method and release at the end. If lock is already acquired by other thread, the current thread is blocked (until lock is released by the locking thread).
- "synchronized" non-static method acquires lock on the current object i.e. "this". Example:

```
class Account {
    // ...
    public synchronized void deposit(double amount) {
        double newBalance = this.balance + amount;
        this.balance = newBalance;
    }
    public synchronized void withdraw(double amount) {
        double newBalance = this.balance - amount;
        this.balance = newBalance;
    }
}
```

- "synchronized" static method acquires lock on metadata object of the class i.e. MyClass.class. Example:

```
class MyClass {
    private static int field = 0;
    // called by incThread
    public synchronized static void incMethod() {
        field++;
    }
    // called by decThread
    public synchronized static void decMethod() {
        field--;
    }
}
```

- "synchronized" block acquires lock on the given object.

```
// assuming that no method in Account class is synchronized.

// thread1
synchronized(acc) {
    acc.deposit(1000.0);
}

// thread2
synchronized(acc) {
    acc.withdraw(1000.0);
}
```

- Alternatively lock can be acquired using RentrantLock since Java 5.0. Example code:

```
class Example {
    private final ReentrantLock rl = new ReentrantLock();
    public void method() {
        rl.lock();
        try {
            // ...
        }
        finally {
            rl.unlock();
        }
    }
}
```

- Synchronized collections
    - Synchronized collections (e.g. Vector, Hashtable, ...) use synchronized keyword (block/method) to handle race conditions.

Inter-thread communication

- wait()
  - Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
  - The current thread must own this object's monitor i.e. wait() must be called within synchronized block/method.
  - The thread releases ownership of this monitor and waits until another thread notifies.
  - The thread then waits until it can re-obtain ownership of the monitor and resumes execution.
- notify()
  - Wakes up a single thread that is waiting on this object's monitor.
  - If multiple threads are waiting on this object, one of them is chosen to be awakened arbitrarily.
  - The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
  - This method should only be called by a thread that is the owner of this object's monitor.
- notifyAll()
  - Wakes up all threads that are waiting on this object's monitor.
  - The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.
  - This method should only be called by a thread that is the owner of this object's monitor.

Member/Nested classes

- By default all Java classes are top-level.
- In Java, classes can be written inside another class/method. They are Member classes.
- Four types of member/nested classes
  - Static member classes -- demo11_01
  - Non-static member class -- demo11_02
  - Local class -- demo11_03
  - Annoymous Inner class -- demo11_04
- When .java file is compiled, separate .class file created for outer class as well as inner class.

Static member classes

- Like other static members of the class (belong to the class, not the object).
- Accessed using outer class (Doesn't need the object of outer class).
- Can access static (private/public) members of the outer class directly.
- Static member class cannot access non-static members of outer class directly.

- The outer class can access all members (including private) of inner class directly (no need of getter/setter).
- The static member classes can be private, public, protected, or default.

```
class Outer {
    private int nonStaticField = 10;
    private static int staticField = 20;

    public static class Inner {
        public void display() {
            System.out.println("Outer.nonStaticField = " + nonStaticField); // error
            System.out.println("Outer.staticField = " + staticField); // ok - 20
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner();
        obj.display();
    }
}
```

Non-static member classes/Inner classes

- Like other non-static members of the class (belong to the object/instance of Outer class).
- Accessed using outer class object (Object of outer class is MUST).
- Can access static & non-static (private) members of the outer class directly.
- The outer class can access all members (including private) of inner class directly (no need of getter/setter).
- The non-static member classes can be private, public, protected, or default.

```
class Outer {
    private int nonStaticField = 10;
    private static int staticField = 20;
    public class Inner {
        public void display() {
            System.out.println("Outer.nonStaticField = " + nonStaticField); // ok-10
            System.out.println("Outer.staticField = " + staticField); // ok-20
        }
    }
}
public class Main {
    public static void main(String[] args) {
        //Outer.Inner obj = new Outer.Inner(); // compiler error
        // create object of inner class
            //Outer outObj = new Outer();
            //Outer.Inner obj = outObj.new Inner();
        Outer.Inner obj = new Outer().new Inner();
        obj.display();
    }
}
```

- If Inner class member has same name as of outer class member, it shadows (hides) the outer class member. Such Outer class members can be accessed explicitly using `Outer.this`.

Static member class and Non-static member class -- Application

```
// top-level class
class LinkedList {
    // static member class
    static class Node {
        private int data;
        private Node next;
        // ...
```

```
    }
    private Node head;
    // non-static member class
    class Iterator {
        private Node trav;
        // ...
    }
    // ...
    public void display() {
        Node trav = head;
        while(trav != null) {
            System.out.println(trav.data);
            trav = trav.next;
        }
    }
}
```

Local class

- Like local variables of a method.
- The class scope is limited to the enclosing method.
- If enclosed in static method, behaves like static member class. If enclosed in non-static method, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.
- We can create any number of objects of local classes within the enclosing method.

```
public class Main {
    private int nonStaticField = 10;
    private static int staticField = 20;
    public static void main(String[] args) {
        final int localVar1 = 1;
        int localVar2 = 2;
        int localVar3 = 3;
        localVar3++;
```

```java
        // local class (in static method) -- behave like static member class
        class Inner {
            public void display() {
                System.out.println("Outer.nonStaticField = " + nonStaticField); // error
                System.out.println("Outer.staticField = " + staticField); // ok 20
                System.out.println("Main.localVar1 = " + localVar1); // ok 1
                System.out.println("Main.localVar2 = " + localVar2); // ok 2
                System.out.println("Main.localVar3 = " + localVar3); // error
            }
        }
        Inner obj = new Inner();
        obj.display();
        //new Inner().display();
    }
}
```

Annoymous Inner class

- Creates a new class inherited from the given class/interface and its object is created.
- If in static context, behaves like static member class. If in non-static context, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.

```java
// (named) local class
class EmpnoComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
}
Arrays.sort(arr, new EmpnoComparator());    // anonymous obj of local class
```

```java
// Anonymous inner class
Comparator<Employee> cmp = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
};
Arrays.sort(arr, cmp);
```