Agenda

- HashTable
- hashcode()
- Map
- Enum
- JVM Architecture
- ~~Java 8 Interfaces~~
- ~~Functional Interfaces~~
- ~~Annoymous Inner Classes~~
- ~~Lambda Expressions~~
- ~~Method references~~
- ~~Stream Programming~~

HashTable Data structure

- Hashtable stores data in key-value pairs so that for the given key, value can be searched in fastest possible time.
- Internally hash-table is a table(array), in which each slot(index) has a bucket(collection).
- Load factor = Number of entries / Number of buckets.
- Multiple keys can compete for the same slot which can cause the collision
- To avoid the collision two techniques are used

1. Open Adderessing
2. Seperate Chaining

- In Seperate Chaning mechanism to avoid the collision Key-value entries are stored in the same bucket depending on hash code of the "key".
- In java we have readymade/ built-in hashtables

1. HashMap
2. LinkedHashMap
3. TreeMap
4. HashTable (Legacy)
5. Properties (Legacy)

- Here we neeed to calculate the hash value of the key using hash function(Override hashcode method).
- The slot in the table is calculated internaly by slot = key.hashcode()%size
- Examples
  - Key=pincode, Value=city/area
  - Key=Employee, Value=Manager
  - Key=Department, Value=list of Employees

hashCode() method

- Object class has hashCode() method, that returns a unique number for each object (by converting its address into a number).
- To use any hash-based data structure hashCode() and equals() method must be implemented.
- If two distinct objects yield same hashCode(), it is referred as collision. More collisions reduce performance.
- Most common technique is to multiply field values with prime numbers to get uniform distribution and lesser collsions.
- hashCode() overriding rules
  - hash code should be calculated on the fields that decides equality of the object.
  - hashCode() should return same hash code each time unless object state is modified.
  - If two objects are equal (by equals()), then their hash code must be same.
  - If two objects are not equal (by equals()), then their hash code may be same (but reduce performance).

Map interface

- Collection of key-value entries (Duplicate "keys" not allowed).
- Implementations: HashMap, LinkedHashMap, TreeMap, Hashtable, ...
- The data can be accessed as set of keys, collection of values, and/or set of key-value entries.
- Map.Entry<K,V> is nested interface of Map<K,V>.
  - K getKey()
  - V getValue()
  - V setValue(V value)
- Abstract methods

```
* boolean isEmpty()
* int size()
* V put(K key, V value)
* V get(Object key)
* Set<K> keySet()
* Collection<V> values()
* Set<Map.Entry<K,V>> entrySet()
* boolean containsValue(Object value)
* boolean containsKey(Object key)
* V remove(Object key)
* void clear()
* void putAll(Map<? extends K,? extends V> map)
```

- Maps not considered as true collection, because it is not inherited from Collection interface.

HashMap class

- Non-ordered map (entries stored in any order -- as per hash code of key)
- Keys must implement equals() and hashCode()
- Fast execution
- Mostly used Map implementation

LinkedHashMap class

- Ordered map (preserves order of insertion)
- Keys must implement equals() and hashCode()
- Slower than HashSet
- Since Java 1.4

TreeMap class

- Sorted navigable map (stores entries in sorted order of key)
- Keys must implement Comparable or provide Comparator
- Slower than HashMap and LinkedHashMap
- Internally based on Red-Black tree.
- Doesn't allow null key (allows null value though).

Hashtable class

- Similar to HashMap class.
- Legacy collection class (since Java 1.0), modified for collection framework (Map interface).
- Synchronized collection -- Thread safe but slower performance
- Inherited from java.util.Dictionary abstract class (it is Obsolete).

Similarity between Set and Map

- Set in internally using map implementation where it have all the values as null.
- In set the the elements are stored as keys and the corresponsing values are null.
- HashSet = HashMap<K,null>
- LinkedHashSet = LinkedHashMap<K,null>
- TreeSet = TreeMap<K,null>
- in set duplicate elements are not allowed, in map duplicate keys are not allowed
- For HashSet,Hashmap, LinkedHashSet, LinkedHashMap duplication is based on equals() and hashcode() of key
- For TreeSet and TreeMap the duplication is based on comparable of K or Comparator of K given in constructor

Enum

- In C enums were internally integers
- In java, It is a keyword added in java 5 and enums are object in java.
- used to make constants for code readability
- mostly used for switch cases
- In java, enums cannot be declared locally (within a method).
- The declared enum is converted into enum class.
- The enum type declared is implicitly inherited from java.lang.Enum class. So it cannot be extended from another class, but enum may implement interfaces.
- The enum constants declared in enum are public static final fields of generated class.
- Enum objects cannot be created explicitly (as generated constructor is private).
- The enums constants can be used in switch-case and can also be compared using == operator.

- The enum may have fields and methods.

```java
public abstract class Enum<E> implements java.lang.Comparable<E>,
java.io.Serializable {
    private final String name;
    private final int ordinal;

    protected Enum(String,int); // sole constructor - can be called from user-
defined enum class only

    public final String name(); // name of enum const
    public final int ordinal(); // position of enum const (0-based)
    public String toString(); // returns name of const
    public final int compareTo(E);// compares with another enum of same type on
basis of ordinal number
    public static <T> T valueOf(Class<T>, String);
    // ...
}
```

```java
// user-defined enum
enum ArithmeticOperations {
    ADDITION, SUBTRACTION, MULIPLICATION, DIVISION
}

// generated enum code
final class ArithmeticOperations extends Enum {

private ArithmeticOperations(String name, int ordinal) {
    super(name, ordinal); // invoke sole constructor Enum(String,int);
}

public static ArithmeticOperations[] values() {
    return (ArithmeticOperations[])$VALUES.clone();
}

public static ArithmeticOperations valueOf(String s) {
    return (ArithmeticOperations)Enum.valueOf(ArithmeticOperations,s);
}


  public static final ArithmeticOperations ADDITION;
  public static final ArithmeticOperations SUBTRACTION;
  public static final ArithmeticOperations MULIPLICATION;
  public static final ArithmeticOperations DIVISION;
  private static final ArithmeticOperations $VALUES[];

  static {
    ADDITION = new ArithmeticOperations("ADDITION", 0);
    SUBTRACTION = new ArithmeticOperations("SUBTRACTION", 1);
    MULIPLICATION = new ArithmeticOperations("MULIPLICATION", 2);
```

```
    DIVISION = new ArithmeticOperations("DIVISION", 3);
    $VALUES = (new ArithmeticOperations[] {
      ADDITION, SUBTRACTION, MULIPLICATION, DIVISION
      });
  }
}
```

JVM Archicecture

- 1. Compilation
  - .class file is cretaed which consists of byte code
- 2. Byte Code
  - It is a machine level instructions that gets executed by the JVM
  - JVM converts byte code into target machine/native code
- 3. Execution
  - java is a tool used to execute the .class file.
  - It loads the .class file and invokes jvm for executing the file from the classpath
- JVM Archicetures Overview
  - ClassLoader + Memory Area + Execution Engine

ClassLoader SubSystem

- It loads and initialize the class

1. Loading

- Three types of classLoaders
  - 1. BootStrap classloader that loads built in java classes from jre/lib jars (rt.jar)
  - 2. Extension classloader that loads the extended classes from jre/lib/ext directory
  - 3. Application classloader that loads the classes from the application classpath
- It reads the classes from the disk and loads into JVM method(memory) area

2. Linking

- Three steps
  - 1. Verifiaction : Bytecode verifier ensures that class is compiled by valid compiler and not tampered
  - 2. Preparation : Memory is allocated for static members and initialized with default values
  - 3. Resolution : Symbolic references in constant pool are replaced by the direct references

3. Initialization

- All static variables of class are assigned with their assigned values(field initializers)
- all static blocks are executed if present

Memory Areas

- Their are 5 memory areas
  - 1. Method Area

- 2. heap Area
- 3. Stack Area
- 4. PC Registers
- 5. Native Method Stack Area

1. Method Area

- Create during JVM startup
- shared by all the threads
- class contents (for all classes) loaded into this area
- Method area also holds constant pool for all loaded classes.

2. Heap Area

- Create during JVM startup
- shared by all the threads
- All allocated objects (with new) are stored in heap
- The string pool is part of heap Area.
- The class Metadata is stored in a java.lang.Class object (in heap) once class is loaded.

3. Stack Area

- Separate stack is created for each thread in JVM (when thread is created).
- When a method is called a new FAR (stack frame) is created on its stack.
- Each stack frame conatins local variable array, operand stack, and other frame data.
- When method returns, the stack frame is destroyed.

4. PC Registers

- Separate PC register is created for each thread.
- It maintains address of the next instruction executed by the thread.
- After an instruction is completed, the address in PC is auto-incremented.

5. Native Method Stack

- Separate native method stack is created for each thread in JVM (when thread is created).
- When a native method is called from the stack, a stack frame is created on its stack.

Execution Engine

- The main component of JVM
- Convert byte code into machine code and execute it (instruction by instruction).
- It consists of
  - 1. Interpreter
  - 2. JIT Compiler
  - 3. Garbage Collector

1. Interpreter

- Each method is interpreted by the interpreter at least once.

- If method is called frequently, interpreting it each time slow down the execution of the program.
- This limitation is overcomed by JIT (added in Java 1.1).

2. JIT compiler

- JIT stands for Just In Time compiler.
- Primary purpose of the JIT compiler to improve the performance.
- If a method is getting invoked multile times, the JIT compiler convert it into native code and cache it.
- If the method is called next time, its cached native code is used to speedup execution process.

3. Profiler

- Tracks resource (memory, threads, ...) utilization for execution.
- Part of JIT that identifies hotspots. It counts number of times any method is executing.
- If the number is more than a threshold value, it is considered as hotspot.

4. Garbage Collector

- When any object is unreferenced, the GC release its memory.

JNI

JNI acts as a bridge between Java method calls and native method implementations.