Hadoop pre-requisites

- Linux
    - Linux commands
    - ssh (password-less login)
- Java
    - Wrapper classes
    - Inheritance (class & interface)
    - Generics
    - Collections (Iterable<>, Map<>, List<>)
    - File IO
    - Reflection

Spark pre-requisites

- Python
- Java
    - Lambda expressions
        - Functional interface
        - Nested/Inner classes
    - Stream operations

Agenda

- Collections -- Map<>
- Inner classes
- Lambda expressions
- Stream operations
- Reflection
- File IO

Collections

- Ordered collection
    - Elements are maintained in some order.
        - The order in which they are inserted.
        - OR In natural order defined by Comparable or Comaparator (sorted order).
- Un-ordered collection
    - Elements order cannot be determined.

Set<>

- Cannot have duplicate elements.
- Implementations:
    - HashSet<>
        - Unordered collection.
        - Very fast.
        - Duplication is detected using hashCode() and equals().
    - LinkedHashSet<>
        - Ordered collection -- order of insertion.
        - Slower than HashSet<>
        - Duplication is detected using hashCode() and equals().
    - TreeSet<>
        - Ordered collection -- natural ordering (sorted order -- using Comparable<> or Comparator<>)
        - Slower than LinkedHashSet<>
        - Duplication is detected based on Comparable<> or Comparator<> as applicable.
            - TreeSet<String> set1 = new TreeSet<String>(); // Comparable of String.
            - TreeSet<String> set2 = new TreeSet<String>(strComparatorObject); // Comparator object passed in constructor

Map<>

- Stores data in key-value pair, so that for a given key value can be searched in fastest possible time.
- Duplicate keys are not allowed.
- Ideal time complexity of searching by key is O(1).
- Java Map<> implementations are based on "chaining" technique i.e. colliding keys will fall into same bucket.

Implementations

- HashMap<>
    - Unordered collection of key-value pairs.
    - Very fast.
    - Duplication of keys is detected using hashCode() and equals().
- LinkedHashMap<>
    - Ordered collection of key-value pairs -- order of insertion.
    - Slower than HashMap<>
    - Duplication of keys is detected using hashCode() and equals().

- TreeMap<>
    - Ordered collection -- natural ordering of keys (sorted order -- using Comparable<> or Comparator<>)
    - Slower than LinkedHashMap<>
- Hashtable<>
    - Same as HashMap<>
    - Legacy collection (before Java 1.2)
    - Synchronized collection and hence slower.

Map<K,V> interface

- V put(K key, V value)
    - add given key-value pair/entry into map.
    - if key is already present, the new value will overwrite the old value and old value will be returned.
    - if key is not already present, the null is returned.
- V get(K key)
    - returns value corresponding to given key, if present.
    - return null if given key is not present.
- int size()
    - returns number of entries in the map.
- V remove(K key)
    - delete entry with given key and return its value.
    - if key is absent, it returns null.
- void clear()
    - delete all entries from the map.
- Set<K> keySet()
    - returns "set" of keys.
- Collection<V> values()
    - return "collection" of values.
- Set<Map.Entry<K,V>> entrySet()
    - The map is collection of key-value entries.
    - Each key-value entry is represented by Map.Entry<> interface.
        - K getKey();
        - V getValue();

Important

- HashMap<> and LinkedHashMap<> follows hashCode() and equals() of **Key** class.
- TreeMap<> follows Comparable of **Key** class.
- When keys are builtin classes (e.g. String, Integer, ...) equals(), hashCode() and Comparable implementation is usually provided in those classes.
- When you are using user-defined class as key, it is programmer's responsibility to implement hashCode(), equals() and/or Comparable in that class.
    - example1: Map<Student,MarkSheet> map = new HashMap<>();
    - example1: Map<Student,List<Integer>> map = new HashMap<>();
- It is recommended to use immutable values as keys. Ensure that you do not modify the key class when entry is done in map.

Nested classes

- The classes defined within another class or method are called as "nested classes".
- Java four types of nested classes
    - static member class
    - non-static member class
    - local class
    - anonymous inner class

Static member class

- defined in another class with static keyword.

```
class Outer {
    static class Inner {
        // ...
    }
}
```

- static member class is like static field/method.
    - it can access only static members of class directly.
    - related to the class (not to particular object of the class).
- The object of static member class can be created, without creating object of outer class.

```
// in main()
Outer.Inner obj = new Outer.Inner();
```

- example:

```
class MyList {
    static class MyNode {
        // ...
    }
    private MyNode head;
    // ...
}
```

Non-Static member class

- defined in another class without static keyword.

```
class Outer {
    class Inner {
        // ...
    }
}
```

- non-static member class is like non-static field/method.
    - it can access static as well as non-static members of class directly.
    - related to the particular object of the class.
- The object of non-static member class cannot be created, without creating object of outer class.

```
Outer objOut = new Outer();
Outer.Inner objIn = obj.new Inner();
```

- The inner class can explicitly access outer class object using "Outer.this". This syntax is typically useful to access outer class member from inner class having same name as of inner class member.
- static and non-static classes can be private, public, protected or default.

```
class MyList {
    static class MyNode {
        // ...
    }
    private MyNode head;
    // ...

    class MyIterator implements Iterator<?> {
        // ...
    }
}
```

Local class

- The class defined in a method (in some class).
- Like a local variable.
    - Cannot be accessed outside the class.
    - It is compiled into .class file named like Outer$1Inner.class.
- It is like static member class, it is defined in a static method.
- It is like non-static member class, it is defined in a non-static method.
- The local class can additionally access final (or effectively final) variables of outer/enclosing method.

Anonymous inner class

- Used to directly create an object of a (un-named) class inherited from given interface/class.
- Can create single object.
- It is like static member class, it is defined in a static method.
- It is like non-static member class, it is defined in a non-static method.
- The inner class can additionally access final (or effectively final) variables of outer/enclosing method.

```
class A {
    // ...
}

// in main()
A obj = new A() {
    // ...
};
```

- In above class, a new class is created inherited from class A and its object is created (referred using obj).
- Using anonymous inner class object, we can access only overridden methods of super-class or super-interface.
- Anonymous inner class is compiled into .class file like Outer$1.class.

Lambda expressions

- Lambda expressions are used in functional programming style to shorten the code.
- In Java8, Lambda expressions are based on Functional interfaces.
- Lambda expression is shorter way to provide implementation for abstract method in relevant functional interface.

```
Collections.sort(list, (e1,e2) -> e1.getEmpno() - e2.getEmpno());
```

- In above example, Lambda expression is implementation of compare() method of Comparator<> interface. Note that 2nd arg of sort() is Comparator<>.

Functional interface

- Contains single abstract method.
- May contain multiple methods with default implementation.
- @FunctionalInterface annotation check whether interface contains only one abstract method during compilation. If more than one abstract method is present, compile time error is raised.
- Java8 comes many general-purpose pre-defined functional interfaces defined in java.util.function package.
    - Consumer<T>
        - void accept(T obj);
    - Predicate<T>
        - boolean test(T obj);

    - Function<T,R>
        - R apply(T obj);
    - Supplier<T>
        - T get();
    - BiFunction<T,U,R>
        - R apply(T o1, U o2);
    - BinaryOperator<T>
        - T apply(T o1, T o2);

Java8 Streams

- Stream interface is added in Java8 in "java.util" package.
- Used for processing elements in any collection in functional programming style.
- Stream represents stream (sequence) of operations to be performed on each element of collection.

Stream characteristics

- Stream is immutable.
    - Performing any operation will result into new stream.
    - The existing elements from collection cannot be modified.
- Streams are evaluated lazily.
    - Stream have two types of operations i.e. intermediate and terminal operations.
    - Intermediate operations produce new stream.
    - Terminal operations produce final result/terminate stream.
    - Stream operations are not performed unless terminal operation is present (at the end of the stream).
- Streams are not reusable.
    - Once operations from stream are completed (due to terminal operation), stream cannot be reused.
    - But we can create a new stream to perform new operations.

Stream operations

Intermediate operations

- Stream<R> map(Function<T,R> fn);
- Stream<T> filter(Predicate<T> fn);
- Stream<R> flatMap(Function<T, Stream<R>> fn);
- Stream<T> sorted(Comparator<T> fn);

Terminal operations

- void forEach(Consumer fn);
- T reduce(T obj, BinaryOperator<T> fn);
- R collect(Collector<T> c);