Core Java

Agenda

- Q & A
- Capturing & Non-capturing Lambda
- Method references
- Java collection framework
  - Introduction
  - Hierarchy
  - List interface and classes
  - equals() method
  - Iterators
  - Collections class
  - Queue Interface

Q & A

- Anonymous class
- Lambda expressions
- Predicate Assignment
- Arithemetic Assignment

Assignments

2. Lambda expression

```
// call countIf() to count number of strings have length more than 6 -- using anonymous inner class
int cnt = countIf(arr, new Predicate<String>() {
    public boolean test(String s) {
        return s.length() > 6;
    }
});
System.out.println("Result: " + cnt); // 2

// call countIf() to count number of strings have length more than 6 -- using lambda expression
cnt = countIf(arr, s -> s.length() > 6);
System.out.println("Result: " + cnt); // 2
```

3. Create a functional `interface Arithmetic` with single abstract method `double calc(double,double)`. Write a static method calculate() in main class as follows. In main(), write a menu driven program that inputs two numbers from the user and calls calculate() method with appropriate lambda expression (in arg3) to perform addition, subtraction, multiplication and division operations.

```
interface Arithmetic {
    double calc(double x,double y);
}
class Main {
    static void calculate(double num1, double num2, Arithmetic op) {
        double result = op.calc(num1, num2);
        System.out.println("Result: "+result);
    }
    static void main(String[] args) {
        double x=12.3, y=2.4;
        calculate(x, y, (a,b) -> a + b);
        calculate(x, y, (a,b) -> a - b);
        calculate(x, y, (a,b) -> a * b);
        calculate(x, y, (a,b) -> a / b);
    }
}
```

javap tool

- Part of JDK.

- To inspect the class byte-code and metadata.
- Usage:
    - javap classname
        - Display public/protected members of the class.
    - javap -p classname
        - Also display private members of the class.
    - javap -v classname
        - Display class metadata, constant pool, byte-code, etc.

Quick Revision

- Member classes
    - Static member class: Inner class object is not dependent on Outer class object.
    - Non-static member class: Inner class object can be created only in context of Outer class object.
    - Local class: Class declared in a method. Scope limited to the method.
    - Anonymous inner class: A class is inherited from given class/interface and its one object is created.

```
ClassName obj = new ClassName() {
    @Override
    public void method() {
        // ...
    }
};
obj.method();
```

```
interface Shape {
    double calcArea();
    double calcPeri();
}

Shape sh = new Shape() {
    private int side = 5;
    public double calcArea() {
        return side * side;
    }
    public double calcPeri() {
        return 4 * side;
    }
};
System.out.println("Area: " + sh.calcArea());
System.out.println("Peri: " + sh.calcPeri());
```

  - The non-static member class, local class and anonymous inner classes are typically designed to create object(s). So it is expected to work with non-static fields/methods in those classes. We cannot declare static fields in these classes. However, we can declare static final field initialized with const value.

```
static int field; // error
final static int field; // error
final static int field = 101; // allowed
```

- Java 8 Interfaces
  - Java 7 -- abstract methods
    - Must be overridden in sub-classes.
  - default methods
    - May be overridden in sub-classes.
    - If not implemented, the default implementation from super interface is considered.
    - May lead to ambiguity (if same default method is inherited to a class from different interfaces)
    - Super-interfaces clash, Super-class wins!
  - static methods
    - static method can be declared interfaces.
    - InterfaceName.method();
  - Functional interface

        - Interface with SAM.
        - @FunctioalInterface -- Check if interface has SAM, else raise error.
    - Lambda expression
        - Short-hand way of implementing SAM in FunctioalInterface.
        - `Predicate<T>` -- SAM -- boolean test(T obj);

```
Predicate<Integer> p = i -> i % 2 == 0;

System.out.println(p.test(12)); // true

System.out.println(p.test(13)); // false
```

    - Method reference

Lambda expressions

Non-capturing lambda expression

- If lambda expression result entirely depends on the arguments passed to it, then it is non-capturing (self-contained).

```
BinaryOperator<Integer> op1 = (a,b) -> a + b;
testMethod(op);
```

```
static void testMethod(BinaryOperator<Integer> op) {
    int x=12, y=5, res;
    res = op.apply(x, y); // res = x + y;
    System.out.println("Result: " + res)
}
```

- In functional programming, such functions/lambdas are referred as pure functions.

Capturing lambda expression

- If lambda expression result also depends on additional variables in the context of the lambda expression passed to it, then it is capturing.

```
int c = 2; // must be effectively final
BinaryOperator<Integer> op = (a,b) -> a + b + c;
testMethod(op);
```

```
static void testMethod(BinaryOperator<Integer> op) {
    int x=12, y=5, res;
    res = op.apply(x, y); // res = x + y + c;
    System.out.println("Result: " + res);
}
```

- Here variable c is bound (captured) into lambda expression. So it can be accessed even out of scope (effectively). Internally it is associated with the method/expression.
- In some functional languages, this is known as Closures.

Method references

- If lambda expression involves single method call, it can be shortened by using method reference.
- Method references are converted into instances of functional interfaces.
- Method reference can be used for class static method, class non-static method, object non-static method or constructor.

Examples

- Class static method: Integer::sum [(a,b) -> Integer.sum(a,b)]
  - Both lambda param passed to static function explicitly

- Class non-static method: String::compareTo [(a,b) -> a.compareTo(b)]
  - First lambda param become implicit param (this) of the function and second is passed explicitly (as arguments).
- Object non-static method: System.out::println [x -> System.out.println(x)]
  - Lambda param is passed to function explicitly.
- Constructor: Date::new [() -> new Date()]
  - Lambda param is passed to constructor explicitly.

Java Collection Framework

Introduction

- Collection framework is Library of reusable data structure classes that is used to develop application.
- Main purpose of collection framework is to manage data/objects in RAM efficiently.
- Collection framework was introduced in Java 1.2 and type-safe implementation is provided in 5.0 (using generics).
- java.util package.
- Java collection framework provides
  - Interfaces -- defines standard methods for the collections.
  - Implementations -- classes that implements various data stuctures.
  - Algorithms -- helper methods like searching, sorting, ...

Collection Hierarchy

- Interfaces: Iterable, Collection, List, Queue, Set, Map, Deque, SortedSet, SortedMap, ...
- Implementations: ArrayList, LinkedList, HashSet, HashMap, ...
- Algorithms: sort(), reverse(), max(), min(), ... -> in Collections class static methods

Iterable interface

- To traverse any collection it provides an Iterator.
- Enable use of for-each loop.
- In java.lang package
- Methods

- Iterator iterator() // SAM
- default Spliterator spliterator()
- default void forEach(Consumer<? super T> action)

Collection interface

- Root interface in collection framework interface hierarchy.
- Most of collection classes are inherited from this interface (indirectly).
- Provides most basic/general functionality for any collection
- Abstract methods
    - boolean add(E e)
    - int size()
    - boolean isEmpty()
    - void clear()
    - boolean contains(Object o)
    - boolean remove(Object o)
    - boolean addAll(Collection<? extends E> c)
    - boolean containsAll(Collection<?> c)
    - boolean removeAll(Collection<?> c)
    - boolean retainAll(Collection<?> c)
    - Object[] toArray()
    - Iterator iterator() -- inherited from Iterable
- Default methods
    - default Stream stream()
    - default Stream parallelStream()
    - default boolean removeIf(Predicate<? super E> filter)

List interface

- Ordered/sequential collection.
- Implementations: ArrayList, Vector, Stack, LinkedList, etc.
- List can contain duplicate elements.

- List can contain multiple null elements.
- Elements can be accessed sequentially (bi-directional using Iterator) or randomly (index based).
- Abstract methods
    - void add(int index, E element)
    - String toString()
    - E get(int index)
    - E set(int index, E element)
    - int indexOf(Object o)
    - int lastIndexOf(Object o)
    - E remove(int index)
    - boolean addAll(int index, Collection<? extends E> c)
    - ListIterator listIterator()
    - ListIterator listIterator(int index)
    - List subList(int fromIndex, int toIndex)
- To store objects of user-defined types in the list, you must override equals() method for the objects. It is mandetory while searching operations like conatins(), indexOf(), lastIndexOf().

ArrayList class

- Internally ArraysList is dynamically growable array.
- Elements can be traversed using Iterator, ListIterator, or using index.
- Default initial capacity of ArrayList is 10. If it gets filled then its capacity gets increased by half of its existing capacity.
- Primary use
    - Random access is very fast
    - Add/remove at the end of list
- Internals (for experts)
    - https://www.javatpoint.com/internal-working-of-arraylist-in-java

Vector class

- Legacy collection class (since Java 1.0), modified for collection framework (List interface).
- Internally Vector is dynamically growable array.

- Elements can be traversed using Enumeration, Iterator, ListIterator, or using index.
- Default initial capacity of vector is 10. If it gets filled then its capacity gets increased/ by its existing capacity.
- Synchronized collection -- Thread safe but slower performance
- Primary use
    - Random access (in multi-threaded applications)
    - Add/remove at the end of list (in multi-threaded applications)

```
NOTE:
* To perform multiple tasks concurrently within a single process, threads are used (thread based multi-tasking or multi-threading).
* When multiple threads are accessing same resource at the same time, the race condition may occur. Due to this undesirable/unexpected results will be produced.
* To avoid this, OS/JVM provides synchronization mechanism. It will provide thread-safe access to the resource (the other threads will be blocked).
```

Iterator vs Enumeration

- Enumeration
    - Since Java 1.0
    - Methods
        - boolean hasMoreElements()
        - E nextElement()
    - Example

```
Enumeration<E> e = v.elements();
while(e.hasMoreElements()) {
    E ele = e.nextElement();
    System.out.println(ele);
}
```

  - Enumeration behaves similar to fail-safe iterator.
- Iterator
  - Part of collection framework (1.2)
  - Methods
    - boolean hasNext()
    - E next()
    - void remove()
  - Example

```
Iterator<E> e = v.iterator();
while(e.hasNext()) {
    E ele = e.next();
    System.out.println(ele);
}
```

- ListIterator
  - Part of collection framework (1.2)
  - Inherited from Iterator
  - Bi-directional access
  - Methods
    - boolean hasNext()
    - E next()
    - int nextIndex()
    - boolean hasPrevious()
    - E previous()
    - int previousIndex()
    - void remove()
    - void set(E e)
    - void add(E e)

Traversal

- Using Iterator

```
Iterator<Integer> itr = list.iterator();
while(itr.hasNext()) {
    Integer i = itr.next();
    System.out.println(i);
}
```

- Using for-each loop

```
for(Integer i:list)
    System.out.println(i);
```

  - Gets converted into Iterator traversal

```
for(Iterator<Integer> itr = list.iterator();itr.hasNext();) {
    Integer i = itr.next();
    System.out.println(i);
}
```

- Traversing List collection

```
for(int i=0; i<list.size(); i++) {
    Integer n = list.get(i);
    System.out.println(n);
}
```

  - Faster for ArrayList/Vector (than Iterator).

    - Much slower for LinkedList.
- forEach() method

```
list.forEach(i -> System.out.println(i));
```

    - Faster than all above approaches.
- Enumeration -- Traversing Vector (Java 1.0)

```
// v is Vector<Integer>
Enumeration<Integer> e = v.elements();
while(e.hasMoreElements()) {
    Integer i = e.nextElement();
    System.out.println(i);
}
```

Fail-fast vs Fail-safe Iterator

- If state of collection is modified (add/remove operation other than iterator methods) while traversing a collection using iterator and iterator methods fails (with ConcurrentModificationException), then iterator is said to be Fail-fast.
    - e.g. Iterators from ArrayList, LinkedList, Vector, ...
- If iterator allows to modify the underlying collection (add/remove operation other than iterator methods) while traversing a collection (NO ConcurrentModificationException), then iterator is said to be Fail-safe.
    - e.g. Iterators from CopyOnWriteArrayList, ...

Synchronized vs Unsynchronized collections

- Synchronized collections are thread-safe and sync checks cause slower execution.
- Legacy collections were synchronized.
    - Vector
    - Stack

    - Hashtable
    - Properties
- Collection classes in collection framework (since 1.2) are non-synchronized (for better performance).
- Collection classes can be converted to synchronized collection using Collections class methods.
    - syncList = Collections.synchronizedList(list)
    - syncSet = Collections.synchronizedSet(set)
    - syncMap = Collections.synchronizedMap(map)

Collections class

- Helper/utility class that provides several static helper methods
- Methods
    - List reverse(List list);
    - List shuffle(List list);
    - void sort(List list, Comparator cmp)
    - E max(Collection list, Comparator cmp);
    - E min(Collection list, Comparator cmp);
    - List synchronizedList(List list);

Collection vs Collections

- Collection interface
    - All methods are public and abstract. They implemented in sub-classes.
    - Since all methods are non-static, must be called on object.

```
Collection<Integer> list = new ArrayList<>();
//List<Integer> list = new ArrayList<>();
//ArrayList<Integer> list = new ArrayList<>();
list.remove(new Integer(12));
```

- Collections class

  - Helper class that contains all static methods.
  - We never create object of "Collections" class.

  ```
  Collections.methodName(...);
  ```

LinkedList class

- Internally LinkedList is doubly linked list.
- Elements can be traversed using Iterator, ListIterator, or using index.
- Primary use
  - Add/remove elements (anywhere)
  - Less contiguous memory available
- Inherited from List<>, Deque<>.

Assignments

1. Store book details in a library in a list -- ArrayList.
   - Book details: isbn(string), price(double), authorName(string), quantity(int)
   - Write a menu driven program to
     1. Add new book in list
     2. Display all books in forward order
     3. Display all books in reverse order
     4. Search a book with given isbn (hint - indexOf())
     5. Delete a book at given index.
     6. Sort all books by price in desc order
     7. Replace book at given index with a new book (input from user)
     8. Remove all books with price < 200. (hint - removeIf())
2. Create a list of strings. Find the string with highest length using Collections.max(). Use lambda expression.

Optional Assignments

1. Create POJO classes User (id, firstName, lastName, email, mobile, passwd) and Quote (id, author, quote, userId). Create a class DbUtil that holds `List<User>` and `List<Quote>` as static members. Implement a class UserDao and QuoteDao as follows. Note that these classes doesn't scan or print anything on terminal/console. Then create UserService and QuoteService to scan data from user and interact with Dao classes. Finally in main() create menu driven program to invoke methods from service classes.

```
public class QuotesDao {
    public Quote findById(int quoteId) {

    }

    public List<Quote> findByUserId(int userId) {

    }

    public List<Quote> findAll() {

    }

    public void addQuote(Quote q) {

    }

    public void updateQuote(Quote q) {
        // replace new quote on index of quote of given id (q.id)
    }
}
```

```
public class UserDao {
    public User findById(int userId) {

    }
    public User findByEmail(String email) {

    }

    public User findByEmailAndPassword(String email, String password) {

    }

    public void addUser(User u) {

    }

    public void updateUser(User u) {

    }
}
```

```
public class UserService {

    public void signIn() {
        // use UserDao to find user with email and password.
        //QuotesApp.user = user;
    }

    public void signUp() {

    }

    public void changePassword() {

    }

    public void changeProfile() {
    }
}
```

```
public void displayUserQuotes() {
    // input user id
}

public void displayAllQuotes() {

}

public void addNewQuote() {
    // call acceptNewQuote() to get new quote and then use QuoteDao to add it
}

public void editQuote() {
    // call acceptModifiedQuote() to get new quote and then use QuoteDao to update it
}

public void deleteQuote() {
    // input quote id and then delete quote by id
}

public void acceptModifiedQuote(Quote quote) {

}

public void acceptNewQuote(Quote quote) {

}
```