Core Java

Day 14 Agenda

- Java 8 Interfaces
  - Default methods
  - Functional interfaces
- Lambda expressions
- Method interfaces
- Stream programming

Java 8 Interfaces

Default methods

- Java 8 allows default methods in interfaces. If method is not overridden, its default implementation in interface is considered.
- This allows adding new functionalities into existing interfaces without breaking old implementations e.g. Collection, Comparator, …

```
interface Emp {
    double getSal();
    default double calcIncentives() {
        return 0.0;
    }
}
class Manager implements Emp {
    // ...
    // calcIncentives() is overridden
    double calcIncentives() {
        return getSal() * 0.2;
    }
}
class Clerk implements Emp {
```

```
    // ...
    // calcIncentives() is not overridden -- so method of interface is considered
}
```

```
new Manager().calcIncentives(); // return sal * 0.2
new Clerk().calcIncentives(); // return 0.0
```

- However default methods will lead to ambiguity errors as well, if same default method is available from multiple interfaces. Error: Duplicate method while declaring class.
- Superclass same method get higher priority. But super-interfaces same method will lead to error.
    - Super-class wins! Super-interfaces clash!!

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class FirstClass implements Displayable, Printable { // compiler error: duplicate method
    // ...
}
class Main {
    public static void main(String[] args) {
        FirstClass obj = new FirstClass();
        obj.show();
    }
}
```

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class Superclass {
    public void show() {
        System.out.println("Superclass.show() called");
    }
}
class SecondClass extends Superclass implements Displayable, Printable {
    // ...
}
class Main {
    public static void main(String[] args) {
        SecondClass obj = new SecondClass();
        obj.show(); // Superclass.show() called
    }
}
```

- A class can invoke methods of super interfaces using InterfaceName.super.

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
```

```
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class FourthClass implements Displayable, Printable {
    @Override
    public  void show() {
        System.out.println("FourthClass.show() called");
        Displayable.super.show();
        Printable.super.show();
    }
}
class Main {
    public static void main(String[] args) {
        FourthClass obj = new FourthClass();
        obj.show(); // calls FourthClass method
    }
}
```

Functional Interface

- If interface contains exactly one abstract method (SAM), it is said to be functional interface.
- It may contain additional default & static methods. E.g. Comparator, Runnable, …
- @FunctionalInterface annotation does compile time check, whether interface contains single abstract method. If not, raise compile time error.

```
@FunctionalInterface    // okay
interface Foo {
    void foo();     // SAM
}
```

```
@FunctionalInterface    // okay
interface FooBar1 {
    void foo();     // SAM
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface    // compiler error
interface FooBar2 {
    void foo();     // AM
    void bar();     // AM
}
```

```
@FunctionalInterface    // compiler error
interface FooBar3 {
    default void foo() {
        /*... */
    }
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface    // okay
interface FooBar4 {
    void foo();         // SAM
    public static void bar() {
        /*... */
```

```
    }
}
```

- Functional interfaces forms foundation for Java lambda expressions and method references.

Built-in functional interfaces

- New set of functional interfaces given in java.util.function package.
  - `Predicate<T>`: test: T -> boolean
  - `Function<T,R>`: apply: T -> R
  - `BiFunction<T,U,R>`: apply: (T,U) -> R
  - `UnaryOperator<T>`: apply: T -> T
  - `BinaryOperator<T>`: apply: (T,T) -> T
  - `Consumer<T>`: accept: T -> void
  - `Supplier<T>`: get: () -> T
- For efficiency primitive type functional interfaces are also supported e.g. IntPredicate, IntConsumer, IntSupplier, IntToDoubleFunction, ToIntFunction, ToIntBiFunction, IntUnaryOperator, IntBinaryOperator.

Lambda expressions

- Traditionally Java uses anonymous inner classes to compact the code. For each inner class separate .class file is created.
- However, code is complex to read and un-efficient to execute.
- Lambda expression is short-hand way of implementing functional interface.
- Its argument types may or may not be given. The types will be inferred.
- Lambda expression can be single liner (expression not statement) or multi-liner block { ... }.

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

```
    }
});
```

```
// Lambda expression -- multi-liner
Arrays.sort(arr, (Emp e1, Emp e2) -> {
    int diff = e1.getEmpno() - e2.getEmpno();
    return diff;
});
```

```
// Lambda expression -- multi-liner -- Argument types inferred
Arrays.sort(arr, (e1, e2) -> {
    int diff = e1.getEmpno() - e2.getEmpno();
    return diff;
});
```

```
// Lambda expression -- single-liner -- with block { ... }
Arrays.sort(arr, (e1, e2) -> {
    return e1.getEmpno() - e2.getEmpno();
});
```

```
// Lambda expression -- single-liner
Arrays.sort(arr, (e1,e2) -> e1.getEmpno() - e2.getEmpno());
```

- Practically lambda expressions are used to pass as argument to various functions.
- Lambda expression enable developers to write concise code (single liners recommended).

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.

Java 8 Streams

- Java 8 Stream is NOT IO streams.
- java.util.stream package.
- Streams follow functional programming model in Java 8.
- The functional programming is based on functional interface (SAM).
- Number of predefined functional interfaces added in Java 8. e.g. Consumer, Supplier, Function, Predicate, ...
- Lambda expression is short-hand way of implementing SAM -- arg types & return type are inferred.
- Java streams represents pipeline of operations through which data is processed.
- Stream operations are of two types
    - Intermediate operations: Yields another stream.
        - filter()
        - map(), flatMap()
        - limit(), skip()
        - sorted(), distinct()
    - Terminal operations: Yields some result.
        - reduce()
        - forEach()
        - collect(), toArray()
        - count(), max(), min()
    - Stream operations are higher order functions (take functional interfaces as arg).

Java stream characteristics

- No storage: Stream is an abstraction. Stream doesn't store the data elements. They are stored in source collection or produced at runtime.
- Immutable: Any operation doesn't change the stream itself. The operations produce new stream of results.
- Lazy evaluation: Stream is evaluated only if they have terminal operation. If terminal operation is not given, stream is not processed.

- Not reusable: Streams processed once (terminal operation) cannot be processed again.

Stream creation

- Collection interface: stream() or parallelStream()
- Arrays class: Arrays.stream()
- Stream interface: static of() method
- Stream interface: static generate() method
- Stream interface: static iterate() method
- Stream interface: static empty() method
- nio Files class: `static Stream<String> lines(filePath)` method

Stream creation

- Collection interface: stream() or parallelStream()

```
List<String> list = new ArrayList<>();
// ...
Stream<String> strm = list.stream();
```

- Arrays class: Arrays.stream()
- Stream interface: static of() method

```
Stream<Integer> strm = Stream.of(arr);
```

- Stream interface: static generate() method
  - generate() internally calls given Supplier in an infinite loop to produce infinite stream of elements.

```
Stream<Double> strm = Stream.generate(() -> Math.random()).limit(25);
```

```
Random r = new Random();
Stream<Integer> strm = Stream.generate(() -> r.nextInt(1000)).limit(10);
```

- Stream interface: static iterate() method
  - iterate() start the stream from given (arg1) "seed" and calls the given UnaryOperator in infinite loop to produce infinite stream of elements.

```
Stream<Integer> strm = Stream.iterate(1, i -> i + 1).limit(10);
```

- Stream interface: static empty() method
- nio Files class: static Stream lines(filePath) method

Stream operations

- Source of elements

```
String[] names = {"Smita", "Rahul", "Rachana", "Amit", "Shraddha", "Nilesh", "Rohan", "Pradnya", "Rohan",
"Pooja", "Lalita"};
```

- Create Stream and display all names

```
Stream.of(names)
    .forEach(s -> System.out.println(s));
```

- filter() -- Get all names ending with "a"
  - `Predicate<T>`: (T) -> boolean

```
Stream.of(names)
    .filter(s -> s.endsWith("a"))
    .forEach(s -> System.out.println(s));
```

- map() -- Convert all names into upper case
  - `Function<T,R>`: (T) -> R

```
Stream.of(names)
    .map(s -> s.toUpperCase())
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in ascending order
  - String class natural ordering is ascending order.
  - sorted() is a stateful operation (i.e. needs all element to sort).

```
Stream.of(names)
    .sorted()
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in descending order
  - `Comparator<T>`: (T,T) -> int

```
Stream.of(names)
    .sorted((x,y) -> y.compareTo(x))
    .forEach(s -> System.out.println(s));
```

- skip() & limit() -- leave first 2 names and print next 4 names

```
Stream.of(names)
    .skip(2)
    .limit(4)
    .forEach(s -> System.out.println(s));
```

- distinct() -- remove duplicate names
  - duplicates are removed according to equals().

```
Stream.of(names)
    .distinct()
    .forEach(s -> System.out.println(s));
```

- count() -- count number of names
  - terminal operation: returns long.

```
long cnt = Stream.of(names)
    .count();
System.out.println(cnt);
```

- collect() -- collects all stream elements into an collection (list, set, or map)

```
List<String> list = Stream.of(names)
        .collect(Collectors.toList());
// Collectors.toList() returns a Collector that can collect all stream elements into a list
```

```

```

```
Set<String> set = Stream.of(names)
    .collect(Collectors.toSet());
// Collectors.toSet() returns a Collector that can collect all stream elements into a set
```

- reduce() -- addition of 1 to 5 numbers

```
int result = Stream
    .iterate(1, i -> i+1)
    .limit(5)
    .reduce(0, (x,y) -> x + y);
```

- max() -- find the max string
  - terminal operation
  - See examples.

Optional<> type

- Few stream operations yield Optional<> value.
- Optional value is a wrapper/box for object of T type or no value.
- It is safer way to deal with null values.
- Get value in the Optional<>:
  - optValue = opt.get();
  - optValue = opt.orElse(defValue);
- Consuming Optional<> value:
  - opt.isPresent() --> boolean;
  - opt.ifPresent(consumer);

Collect Stream result

- Collecting stream result is terminal operation.

- Object[] toArrray()
- R collect(Collector)
    - Collectors.toList(), Collectors.toSet(), Collectors.toCollection(), Collectors.joining()
    - Collectors.toMap(key, value)

Stream of primitive types

- Efficient in terms of storage and processing. No auto-boxing and unboxing is done.
- IntStream class
    - IntStream.of() or IntStream.range() or IntStream.rangeClosed() or Random.ints()
    - sum(), min(), max(), average(), summaryStatistics(),
    - OptionalInt reduce().

Assignment

1. Create an interface Emp with abstract method `double getSal()` and a default method `default double calcIncentives()`. The default method simply returns 0.0. Create a class Manager (with fields basicSalary and dearanceAllowance) inherited from Emp. In this class override getSal() method (basicSalary + dearanceAllowance) as well as calcIncentives() method (20% of basicSalary). Create another class Labor (with fields hours and rate) inherited from Emp interface. In this class override getSal() method (hours * rate) as well as calcIncentives() method (5% of salary if hours > 300, otherwise no incentives). Create another class Clerk (with field salary) inherited from Emp interface. In this class override getSal() method (salary). Do not override, calcIncentives() in Clerk class. In Emp interface create a static method `static double calcTotalIncome(Emp arr[])` that calculate total income (salary + incentives) of all employees in the given array.

2. Use following method to count number of strings with length > 6 in given array.

```
public static int countIf(String[] arr, Predicate<String> cond) {
    int count = 0;
    for(String str: arr) {
        if(cond.test(str))
            count++;
    }
    return count;
}
```

```
public static void main(String[] args) {
    String[] arr = { "Nilesh", "Shubham", "Pratik", "Omkar", "Prashant" };
    // 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 expressions
}
```

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.

```
static void calculate(double num1, double num2, Arithmetic op) {
    double result = op.calc(num1, num2);
    System.out.println("Result: "+result);
}
```

4. Create a functional `interface Check<T>` with single abstract method `boolean compare(T x, T y)`. Create a static method in main class to test array elements `static <T> int countIf(T[] arr, T key, Check<T> c)`. This method should return count of elements in the array for which given check is satisfied. The check will be given as lambda expression. Example call to countIf() from main() will be as follows.

```
Integer [] arr = {44, 77, 99, 22, 55, 66};
Integer key = 50;
```

```
int cnt = countIf(arr, key, (x,y)-> x > y);
System.out.println("Count = " + cnt); // 4 (because 4 elements in array are greater than given key i.e. 50)
```

5. In above assignment, create one more array of Double (constant values) where few elements are repeated. Input a key from user and check how many times key is repeated in the array using appropriate lambda expression.
6. Calculate the factorial of the given number using stream operations.
7. Write a program to calculate sum of 10 random integers using streams.
8. Create an IntStream to represent numbers from 1 to 10. Call various functions like sum(), summaryStatistics() and observe the output.