Core Java

Agenda

- Quick Revision
- Member/Nested classes
- Java 8 Interfaces
    - Default methods
    - Static methods
    - Functional Interface
    - Built-in functional interfaces
    - Lambda expressions
    - Method references

Quick Revision

- Generic programming enable us to write common code for different data types.
- Generic classes, interfaces and methods.
- Generic method

```
public static <T extends Number> T findMin(T[] arr) {
    T min = arr[0];
    for(T ele:arr) {
        if(ele.doubleValue() < min.doubleValue())
            min = ele;
    }
    return min;
}
```

- For generic classses/interfaces generic type needs to be given while instantiating it; while for generic methods type is inferred automatically.
- Java generics work with reference types only. Doesn't work with primitive types.
- Type erasure: The generic type information is erased (not maintained) at runtime (in JVM). Box and Box both are internally (JVM) treated as Box objects. The field "T obj" in Box class, is treated as "Object obj".
  - Because of this method overloading with genric type difference is not allowed.

    ```
    void printBox(Box<Integer> b) { ... }
        // void printBox(Box b) { ... } <-- In JVM
    void printBox(Box<Double> b) { ... } //compiler error
        // void printBox(Box b) { ... } <-- In JVM
    ```
- Generics provides type-safety i.e. compile time type checking.
- Java generics are most used with data structures and algorithms (Java Collections).
- Comparable and Comparator are generic interfaces that provide standard way of comparing two objects.

```
class Rectangle implements Comparable<Rectangle> {
    private double length, breadth;
    // ...
    public double calcArea() {
        return length * breadth;
    }
    public double calcPeri() {
        return 2 * (length + breadth);
    }
    public int compareTo(Rectangle other) {
        int diff = (int)Math.signum(this.calcArea() - other.calcArea());
        return diff;
```

```
    }
}
```

```
Arrays.sort(arr); // internally comparison will be done using compareTo() of Rectangle
```

```
class RectPeriComparator implements Comparator<Rectangle> {
    public int compare(Rectangle r1, Rectangle r2) {
        int diff = (int)Math.signum(r1.calcPeri() - r2.calcPeri());
        return diff;
    }
}

Arrays.sort(arr, new RectPeriComparator()); // internally comparison will be done using compare() of RectPeriComparator
```

Assignment

```
static <T> selectionSort(T[] arr, Comparator<T> c) {
    for(int i=0; i<arr.length; i++) {
        for(int j=i+1; j<arr.length; j++) {
            // arr[i] > arr[j]
            if(c.compare(arr[i], arr[j]) > 0) {
                T temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

```
Integer[] arr = {44, 77, 22 ,55, 33};
class IntDescComparator implements Comparator<Integer> {
    public int compare(Integer x, Integer y) {
        return y - x;
    }
}
selectionSort(arr, new IntDescComparator());
// print array
```

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
  - Non-static member class
  - Local class
  - Annoymous Inner class
- 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++;
        // 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.

```
// (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
```

```
// 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);
```

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

Java 8 Interfaces

- Before Java 8
    - Interfaces are used to design specification/standards. It contains only declarations – public abstract.

    ```
    interface Geometry {
        /*public static final*/ double PI = 3.14;
        /*public abstract*/ int calcRectArea(int length, int breadth);
        /*public abstract*/ int calcRectPeri(int length, int breadth);
    }
    ```

    - As interfaces doesn't contain method implementations, multiple interface inheritance is supported (no ambiguity error).
    - Interfaces are immutable. One should not modify interface once published.

- Java 8 added many new features in interfaces in order to support functional programming in Java. Many of these features also contradicts earlier Java/OOP concepts.

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, ...

```java
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
}
```

```java
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 {
```

```java
    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.

```java
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
    }
}
```

Static methods

- Before Java 8, interfaces allowed public static final fields.
- Java 8 also allows the static methods in interfaces.
- They act as helper methods and thus eliminates need of helper classes like Collections, ...

```
interface Emp {
    double getSal();
    public static double calcTotalSalary(Emp[] a) {
        double total = 0.0;
        for(int i=0; i<a.length; i++)
            total += a[i].getSal();
        return total;
    }
}
```

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.

```java
@FunctionalInterface // okay
interface Foo {
    void foo();
}
@FunctionalInterface // okay
interface FooBar1 {
    void foo();
    default void bar() {
        /*... */
    }
}
@FunctionalInterface // compiler error
interface FooBar2 {
    void foo();
    void bar();
}
@FunctionalInterface // compiler error
interface FooBar3 {
    default void foo() {
        /*... */
    }
    default 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: test: T -> boolean
  - Function<T,R>: apply: T -> R
  - BiFunction<T,U,R>: apply: (T,U) -> R
  - UnaryOperator: apply: T -> T
  - BinaryOperator: apply: (T,T) -> T
  - Consumer: accept: T -> void
  - Supplier: 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 { ... }.

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

```java
// 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;
```

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;
BinaryOperator<Integer> op2 = (a,b) -> a + b + c;
```

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

Assignment

- 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 Labor (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(Employee arr[])` that calculate total income (salary + incentives) of all employees in the given array.

- 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;
}
```

```java
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
}
```