Core Java

- Annoymous Inner classes
- Java 8 Interfaces
  - default methods
  - static methods
  - functional interfaces
- Lambda expressions
- Stream Programming

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, ...

```
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 {
```

```java
    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, ...

```java
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(); // SAM
  }
  ```

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

  ```java
  @FunctionalInterface // NO -- error
  interface FooBar2 {
      void foo(); // AM
      void bar(); // AM
  }
  ```

  ```java
  @FunctionalInterface // NO -- error
  interface FooBar3 {
      default void foo() {
          /*... */
      }
      default void bar() {
          /*... */
      }
  }
  ```

  ```java
  @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);
```

```java
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.