Agenda

- Date Parsing
- Java 8 Interfaces
  - default Methods
  - Static Methods
- Functional Interfaces
- Annoymous Inner Classes
- Lambda Expressions
- Method references

Java DateTime APIs

- DateTime APIs till Java 7

```
// java.util.Date
Date d = new Date();
System.out.println("Timestamp: " + d.getTime());
// number of milliseconds since 1-1-1970 00:00.
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
System.out.println("Date: " + sdf.format(d));

// java.util.Date
String str = "28-09-1983";
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
Date d = sdf.parse(str);
System.out.println(d.toString());

// java.util.Calendar
Calendar c = Calendar.getInstance();
System.out.println(c.toString());
System.out.println("Current Year: " + calendar.get(Calendar.YEAR));
System.out.println("Current Month: " + calendar.get(Calendar.MONTH));
System.out.println("Current Date: " + calendar.get(Calendar.DATE));
```

- Limitations of existing DateTime APIs
  - Thread safety
  - API design and ease of understanding
  - ZonedDate and Time
- Most commonly used java 8 onwards new classes are LocalDate, LocalTime and LocalDateTime.
- LocalDate

```
LocalDate localDate = LocalDate.now();
LocalDate tomorrow = localDate.plusDays(1);
DayOfWeek day = tomorrow.getDayOfWeek();
```

```
int date = tomorrow.getDayOfMonth();
System.out.println("Date: " +
tomorrow.format(DateTimeFormatter.ofPattern("dd-MMM-yyyy")));

//LocalDate date = LocalDate.of(1983, 09, 28);
LocalDate date = LocalDate.parse("1983-09-28");
System.out.println("Is Leap Year: " + date.isLeapYear());
```

- LocalTime

```
LocalTime now = LocalTime.now();
LocalTime nextHour = now.plus(1, ChronoUnit.HOURS);
System.out.println("Hour: " + nextHour.getHour());
System.out.println("Time: " +
nextHour.format(DateTimeFormatter.ofPattern("HH:mm")));
```

- LocalDateTime

```
LocalDateTime now = LocalDateTime.now();
LocalDateTime dt = LocalDateTime.parse("2000-01-30T06:30:00");
dt.minusHours(2);
System.out.println(dt.toString());
```

Java 8 Interface

- 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.
- From Java 8 onwards we can now also add the method body in the interface, however the methods should be either default or static

1. 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");
    }
}
```

2. 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 Interfaces

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

```
@FunctionalInterface // NO -- 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.

Annonymous 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();
    }
});
```

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.

Method references

- lambda expression is an short-hand implementation of Single Abstract Method (Functional Interface)
- Method reference is short-hand of lambda-expression
- 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.