# Functional Interfaces And Lambda Expressions

## Objective

* What are functional interfaces
* Rules for declaring functional interface
* What are Lambda expressions
* Rules for implementing functional interfaces using lambda expressions

## Description

* Functional interfaces are interfaces that can have only 1 abstract method
* For the compiler to enforce this rule these interfaces can be annotated with @FunctionalInterface
* Lambda expressions require an object of type functional interface
* Lambda expressions simplify the code for anonymous inner class implementations.
* To create a functional interface annotate the interface with @FunctionalInterface
* It should consist of only one single abstract method.
* Create an interface **Tester.java** as follows:

|  |
| --- |
| @FunctionalInterface  **public** **interface** Tester {  **public** **boolean** test(T t);  } |

* Create a class **TestTester.java** and provide the implementation for the Tester interface which can be done using anonymous inner class as follows :

|  |
| --- |
| **public** **class** TestTester {  **public** **static** **void** main(String[] args) {  //PRE JAVA 8 ANONYMOUS CLASS FOR Tester  1 Tester<Integer> test = **new** Tester<Integer>() {  2 @Override  3 **public** **boolean** test(Integer t)  4 {  5 **if**(t%2==0)  6 **return** **true**;  7 **return** **false**;  8 }  9};  }  } |
|  |

* Using java 8 Lambda expressions lines 1-9 can be reduced as follows:

|  |
| --- |
| **public** **class** TestTester {  **public** **static** **void** main(String[] args) {  //SINCE JAVA 8 ANONYMOUS CLASS FOR Tester  Tester<Integer> t = (n)->{  **if**(n%2==0)  **return** **true**;  **return** **false**;};  **}**  } |

* To test the above implementor invoke the method test as follows:

|  |
| --- |
| System.***out***.println(test.test(5));  System.***out***.println(t.test(4)); |

* Following class demonstrates different ways of using Lambda expressions on existing classes.

|  |
| --- |
| **public** **class** BasicExamples {    **public** **static** **void** main(String[] args) {  List<String> list = Arrays.*asList*( "a", "e", "d" );  **for**(String s:list)  System.***out***.println(s);  System.***out***.println();    /\*Please notice the type of argument e is being inferred by the  compiler.\*/  Arrays.*asList*( "a", "b", "d" ).forEach( (e) ->  System.***out***.println( e ) );    //OR WITHOUT () AROUND e  Arrays.*asList*( "a", "b", "d" ).forEach(e ->  System.***out***.println( e ) );  //STATIC METHOD REFERENCES  Arrays.*asList*( "a", "b", "d" ).forEach(  System.***out***::println );    /\*You may explicitly provide the type of the parameter,wrapping  the definition in brackets.For example:\*/  System.***out***.println();  Arrays.*asList*( "a", "b", "d" ).forEach( ( String e ) ->  System.***out***.println( e ) );    /\*In case lambda’s body is more complex, it may be wrapped into  curly brackets,as the usual function definition in Java. For  example:\*/  System.***out***.println();  Arrays.*asList*( "a", "b", "d" ).forEach( e -> {  System.***out***.print( e );  System.***out***.print( e );  } );  /\*Lambdas may reference the class members and local variables  (implicitly making them effectively final if they are not).For  example, those two snippets are equivalent:\*/  System.***out***.println();  String separator = ",";  Arrays.*asList*( "a", "b", "d" ).forEach(  ( String e ) -> System.***out***.print( e + separator ) );    //WITHOUT LAMBDA  Collections.*sort*(list, **new** Comparator<String>() {  @Override  **public** **int** compare(String arg0, String arg1) {  **return** arg0.compareTo(arg1);  }  });  //WITH LAMBDA  System.***out***.println();  Collections.*sort*(list, (s1,s2)->{**return**  s1.compareTo(s2);});  list.forEach(s->System.***out***.println(s));  //OR NO NEED TO SPECIFY RETURN IT IS AUTOMATICALLY INFERRED  System.***out***.println();  Collections.*sort*(list, (s1,s2)->s2.compareTo(s1));  //OR USING METHOD REFERENCES COVERED LATER  list.forEach(System.***out***::println);  }  } |
|  |

## Assignment 1

* In Eclipse under Java Perspective create a project **Java8Assign\_01** .
* Under ***com.sum*** package create a functional interface **Validate.java** having single method test() with following signature:
  + Return type Boolean
  + Integer as a single parameter
* Under **com.sum** package create a class **SumOfIntegers.java** as follows:

|  |
| --- |
| **package** com.sum;  **public** **class** SumOfIntegers {  **public** **static** **int** sumAll(List<Integer> numbers, Validate p) {  **int** total = 0;  **for** (**int** number : numbers) {  **if** (p.test(number)) {  total += number;  }  }  **return** total;}  } |

* This method provides the sum of all integers in the list with different conditions

## TO\_DO…….

* Under **com.test** package create a test class with name ***TestSumOfIntegers.java***
* Invoke the sumAll() method from this test class.
* Provide the implementation for Validate method using Lambda expression
* Use the following different conditions while implementing the test() of Validate interface
  + Find the sum of all even integers
  + Find the sum of all integers greater than 3

## Conclusion

* We learned how to create functional interfaces and its syntax
* Lamdba expressions are defined on an object of type functional interface.
* These expressions have a syntax which we llearned in this topic

# Default/Static Methods

## Objective

* What are default/static methods
* Use of default and static keywords in interface declaration

## Description

* Functional interfaces can have only 1 abstract method but more than one default or static methods
* Default methods can be overridden
* Static methods are not overridden
* Create a functional interface **MyData.java** as follows :

|  |
| --- |
| @FunctionalInterface  **public** **interface** MyData {  **public** **void** show();  **default** **void** print(String str) {  **if** (!*isNull*(str))  System.***out***.println("MyData Print::" + str);  }  **static** **boolean** isNull(String str) {  System.***out***.println("Interface Null Check");  **return** str == **null** ? **true** : "".equals(str) ? **true** : **false**;  }  } |

* Create an implementor class **MyDataImpl.java** for above interface MyData. Note the following points:
  + This class needs to provide the implementation of the abstract method
  + Default method implementation is optional
  + Static methods are not overridden

|  |
| --- |
| **public** **class** MyDataImpl **implements** MyData {  @Override  **public** **void** show() {  }  @Override  **public** **void** print(String str) {  }  } |

## Conclusion

* Interfaces having methods declared with default/static keyword are default/static methods

Default/static methods need to have a body within the interface itself

# Method References

## Objective

* What are method references
* Syntax for referring methods and constructors using this shortcut

## Description

* Method refernces are syntactical shortcut for Lambda expressions
* Whereas Lambda expressions are used to provide implementations for functional interfaces
* On the other hand, method refernces are used to refer existing methods or constructors
* Create a class **Car.java** as follows:
* Supplier is a predefined functional interface in java.util.function package

|  |
| --- |
| **import** java.util.function.Supplier;  **public** **class** Car {  **public** String getName() {  **return** name;  }  **public** **static** Car create( **final** Supplier<Car> supplier ) {  **return** supplier.get();  }  //static method  **public** **static** **void** collide( **final** Car car ) {  System.***out***.println( "Collided " + car.toString() );  }  //non static bounded method  **public** **void** follow( **final** Car another ) {  System.***out***.println("Following the "+another.toString());  }  //non static unbounded method  **public** **void** repair() {  System.***out***.println( "Repaired " + **this**.toString() );  }  } |
|  |

* Create a class **CarDemo.java** that creates a collection of car objects and invokes the methods using the method references syntax as follows:

|  |
| --- |
| **public** **class** CarDemo {  **public** **static** **void** main(String[] args) {  //WITHOUT LAMBDA I AM ABLE TO DEFINE get() AND ASK IT TO INVOKE  A CONSTRUCTOR  Car.*create*(**new** Supplier<Car>() {  @Override  **public** Car get() {  // **TODO** Auto-generated method stub  **return** **new** Car();  }  });  //WITH LAMBDA  Car ob = Car.*create*(()->**new** Car());  /\*\*  \* The first type of method references is constructor reference  \* with the syntax Class::new  \* or alternatively, for generics, Class< T >::new.  \* Please notice that the constructor has no arguments.  \*/  **final** Car car = Car.*create*( Car::**new** );  **final** List< Car > cars = Arrays.*asList*(car);  /\*\*  \* The second type is reference to static method with the  syntax  \* Class::static\_method.  \* Please notice that the method accepts exactly one parameter of type Car.  \*/  cars.forEach(Car::*collide*);  /\*\*  \* The third type is reference to instance method of arbitrary object of specific type with the syntax Class::method.  \* Please notice, no arguments are accepted by the method.  \*/  cars.forEach( Car::repair );  /\*\*  \* And the last, fourth type is reference to instance method of particular class instance the syntax instance::method.  \* Please notice that method accepts exactly one parameter of type Car.  \*/  Car police = Car.*create*(Car::**new**);  cars.forEach(police::follow);  }  } |
|  |
|  |

## Assignment

* In Eclipse under Java Perspective create a project **Java8Assign\_02** .
* Under ***com.details*** package create a class **Person.java** as follows:

|  |
| --- |
| **class** Person {  Person() {}  **public** **static** Person create( **final** Supplier<Person> supplier ) {  **return** supplier.get();  }  **public** **static** **void** entered( **final** Person person ) {  System.***out***.println( "Entered " + person.toString() );  }  **public** **void** followed( **final** Person another ) {  System.***out***.println("Following the "+another.toString());  }  **public** **void** exited() {  System.***out***.println( "Exited " + **this**.toString() );  }  } |

## TO\_DO…….

* Under **com.test** package create a class named **PersonDemo.java**
* Create a collection of person objects.
* Invoke the constructor and all the methods of Person class in PersonDemo class using the syntax of method references.

## Conclusion

* Method references can be created using :: keyword
* They can be used to invoke existing methods or default constructors as seen in the demo

# Stream API

## Objective

* What is a Stream interface
* Purpose of streams in java 8 collections and arrays
* Different methods in Stream API, to handle various operations like filter, sort, finding min, max or average on the data in the collection or arrays

## Description

* Stream is a data structure that is computed on-demand as opposed to collections which needs the data populated before using them.
* Streams don’t store data, it just operates on the source data structure and produce pipelined structure.
* Stream operations use functional interfaces.
* Streams internal iteration principle helps in achieving lazy-seeking allowing higher performance and scope for optimization
* Streams are consumable
* Streams support sequential as well as parallel processing
* Streams can be build in various ways from collections, arrays or independently
* Lets have a look at the example on generating streams.
* Create a class **GenerateStreams.java** as follows. It demonstrates ways of generating streams.

|  |
| --- |
| **public** **class** GenerateStreamsDemo {  **public** **static** **void** main(String[] args) {  //Using Stream.of(val1, val2, val3….)  Stream<Integer> streamof = Stream.*of*(1,2,3,4,5,6,7,8,9);  streamof.forEach(p -> System.***out***.println(p));  //Using Stream.of(arrayOfElements)  Stream<Integer> streamofarr = Stream.*of*( **new**  Integer[]{1,2,3,4,5,6,7,8,9} );  streamofarr.forEach(p -> System.***out***.println(p));  //Using someList.stream()  List<Integer> list = **new** ArrayList<Integer>();  **for**(**int** i = 1; i< 10; i++){  list.add(i);  }  Stream<Integer> streamlist = list.stream();  streamlist.forEach(p -> System.***out***.println(p));    //Using Stream.generate() or Stream.iterate() functions  Stream<Date> streamgen = Stream.*generate*(() -> { **return**  **new** Date();});  streamgen.limit(10).forEach(p -> System.***out***.println(p));  //Using String chars or String tokens  IntStream streamchars = "12345\_abcdefg".chars();  streamchars.forEach(p -> System.***out***.println(p));  //OR  System.***out***.println();  Stream<String> stream = Stream.*of*("A$B$C".split("\\$"));    //System.out.println(stream.count());  stream.forEach(p -> System.***out***.println(p));  }  } |

* Stream operations can be divided into two parts
  + Intermediate Operations
  + Terminal Operations
* Before we go ahead with the demonstration of various operations, lets create a collection of Person.
* First, create a class **Person.java** as follows:

|  |
| --- |
| **public** **class** Person **implements** Comparable<Person> {  String name;  **int** age;  List<Address> address;  **public** Person() {  }  **public** Person(String name, **int** age, List<Address> address) {  **super**();  **this**.name = name;  **this**.age = age;  **this**.address = address;  }  **public** List<Address> getAddress() {  **return** address;  }  **public** **void** setAddress(List<Address> address) {  **this**.address = address;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  @Override  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + ",  address=" + address  + "]";  }  @Override  **public** **int** compareTo(Person o) {  **return** name.compareTo(o.name);  }  } |
|  |

* Create class Address as follows:

|  |
| --- |
| **public** **class** Address {  String city;  **public** Address(String city) {  **super**();  **this**.city = city;  }  **public** String getCity() {  **return** city;  }  **public** **void** setCity(String city) {  **this**.city = city;  }  @Override  **public** String toString() {  **return** "City : " + city ;  } } |

* Populate the collection with Person class objects as follows:

|  |
| --- |
| List<Person> person = **new** ArrayList<>();  person.add(**new** Person("Shalini",24,Arrays.*asList*(**new** Address("Mumbai"))));  person.add(**new** Person("Manish",26,Arrays.*asList*(**new** Address("Delhi"))));  person.add(**new** Person("Palash",28,Arrays.*asList*(**new** Address("Mumbai"))));  person.add(**new** Person("Amit",27,Arrays.*asList*(**new** Address("Chennai"))));  person.add(**new** Person("Asha",33,Arrays.*asList*(**new** Address("Pune"))));  person.add(**new** Person("Kshitij",25,Arrays.*asList*(**new** Address("Mumbai"))));  person.add(**new** Person("Ashok",22,Arrays.*asList*(**new** Address("Nagpur"))));  person.add(**new** Person("Pooja",34,Arrays.*asList*(**new** Address("Bhopal"))));  person.add(**new** Person("Deepti",45,Arrays.*asList*(**new** Address("Mumbai"))));  person.add(**new** Person("Ajit",34,Arrays.*asList*(**new** Address("Mumbai")))); |

* This collection will be referred in the following demos of stream operations.

## Intermediate Operations

* Intermediate Operations process over a Stream and return Stream as a response.
* Multiple intermediate operations can be chained in a row.
* Lets demonstrate the various intermediate operations like map,filter, sorted and flatmap.

## Filtering - Stream<T> filter(Predicate<? super T> predicate)

* The filter method is used to filter out elements from a stream, depending upon some condition.
* The filter method accepts a Predicate as an argument.
* A Predicate is a function that returns boolean.
* The filter method returns a stream containing the elements matching to the given predicate
* Example:

|  |
| --- |
| //filter() FUNCTION FILTERS THE NAMES OF PERSON THAT STARTS WITH A  //FINALLY forEach() PRINTS THE NAMES  person.stream().filter(members->members.getName().startsWith("A"))  .forEach(System.***out***::println); |

## Mapping - <R> Stream<R> map(Function<? super T, ? extends R> mapper)

* Process of changing the form of the elements in a stream
* The map is a stream operation which takes another function as an argument.
* The function should take each element of a stream as a parameter and return newly created/modified element as a response.
* The given function is then applied to each element of the stream.
* Example:

|  |
| --- |
| //map FUNCTION IS APPLIED ON STREAM OF OBJECTS  //IT RETURNS A STREAM CONSISTING OF AGE OF EACH STUDENT  //FINALLY forEach() PRINTS THE AGE  person.stream().map(p=>p.getAge())  .forEach(System.***out***::println); |

## Assignment

* In Eclipse under Java Perspective create a project **Java8Assign\_03** .
* Under ***com.weather*** package create a class **City.java** as follows:

|  |
| --- |
| **public** **class** City {  String name;  List<Temperature> temperatures;  **public** City(String name, List<Temperature> temperatures) {  **this**.name = name;  **this**.temperatures = temperatures;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** List<Temperature> getTemperatures() {  **return** temperatures;  }  **public** **void** setTemperatures(List<Temperature> temperatures) {  **this**.temperatures = temperatures;  }  @Override  **public** String toString() {  **return** "City [name=" + name + " :\n temperatures=" + temperatures + "]\n";  }  } |

* Under ***com.weather*** package create a class **Temperature.java** as follows:

|  |
| --- |
| **public** **class** Temperature {  Date date;  BigDecimal reading;  **public** Temperature(Date date, BigDecimal reading) {  **super**();  **this**.date = date;  **this**.reading = reading;  }  **public** Date getDate() {  **return** date;  }  **public** **void** setDate(Date date) {  **this**.date = date;  }  **public** BigDecimal getReading() {  **return** reading;  }  **public** **void** setReading(BigDecimal reading) {  **this**.reading = reading;  }  @Override  **public** String toString() {  **return** "Temperature [date=" + date + ", reading=" + reading + "]\n";  }  } |

**TO\_DO…….**

* Under **com.test** package create a test class with name **TestCity*.java***
* Create objects of city and populate the data in a collection.
* Perform the following operations on this collection using the Stream API methods
  + Generate a stream on this collection and display the list of City along with temperatures reading
  + Generate a stream on this collection and display the list of cities whose name starts with A
  + Generate a stream on this collection and display the names of cities in upper case

## Conclusion

* Stream operate on the data, process them and gives a new stream that can be stored in a collection.
* Various stream methods are available, intermediate operations can be chained and terminal operations finally processes the data and the stream is then consumed.
* Consumed stream cannot be reused.

# Stream API Contd

## Description

## Intermediate Operations Continued….

## Flattening - <R> Stream<R> map(Function<? super T, ? extends R> mapper)

* The flatMap transforms each element of a stream into another form (just like map), and generates sub streams of the newly formed elements.
* Finally, it flattens all of the sub streams into a single stream of elements.
* As the flatMap is a map type of function, it also takes a function and applies (maps) that function to each of the element in the stream.
* The difference between map and flatMap is, the map accepts a function that returns a mapped element and then the map function returns a stream of such elements.
* On the other hand, the flatMap accepts a function that returns streams of the mapped elements and then the flatMap finally returns a collective stream of all of the sub streams that are created by the each execution of the passed function.
* Example:

|  |
| --- |
| //FUNCTION PASSED TO flatMap() RETURNS A STREAM OF ADDRESS OBJECTS  //Finally forEach prints the ADDRESS OF EACH PERSON  Stream<Person> perstream = person.stream();  Stream<Address> addrstream =  perstream.flatMap(p>p.getAddress().stream());  addrstream.forEach(System.***out***::println); |

## Sorting - Stream<T> sorted();

* The sorted() gets a sorted collection in natural order
* Example:

|  |
| --- |
| //sorted FUNCTION SORTS THE PERSON NAMES IN NATURAL ORDERi.e ASCENDING  //Finally forEach PRINTS THE PERSON DETAILS IN ASCENDING ORDER OF THEIR  NAMES  person.stream().sorted().forEach(System.***out***::println); |

## Terminal Operations

* An usual stream operation flow can have a pipe of multiple intermediate operations and a terminal operation at the end.
* The intermediate operations are called upon streams and there return type is stream.Hence, they can be easily chained together in order to get a complete processing pipeline.
* Any of such a stream pipeline must end with a valid Terminal Operation
* Lets demonstrate the various terminal operations like reduce, collect

## Reducing - T reduce(T identity, BinaryOperator<T> accumulator);

* The reducing has a capability of processing the elements in a stream repeatedly to produce an output in the form of a single element. Reducing reduces the entire stream into a single value.
* The reduce operation is very useful for calculating the sum of all elements in the stream or calculating max or min element out of a stream.
* Summing all of the elements in a collection needs the resulting variable set to an initial value (zero), and then combining the result with each element of the collection (result += element).
* The reduce function simplifies this with the help of internal iterations.
* Example:

|  |
| --- |
| //reduce() FINDS THE SUM OF AGES IN PERSON CLASS  Stream<Person> stream = person.stream();  Stream<Integer> age = stream.map(Person::getAge);  Integer sum=age.reduce(0,(p,q)>p+q);  System.***out***.println("Sum of ages "+sum); |

## Collecting - <R, A> R collect(Collector<? super T, A, R> collector);

* Streams offers various methods to operate on the data.
* This processed data can be collected into a collection or array for future purpose as streams do not store data.
* This version of collect() takes an argument of Collectors class which has various methods to accumulate data in collections or arrays.
* Example:

|  |
| --- |
| List<Person> list = person.stream().filter(p->p.getAge()>25)  .collect(Collectors.*toList*());  list.forEach(System.***out***::println); |

## Assignment

* Continue with the assignment given in 4.3 and perform the following operations on the streams of collections:

TO\_DO…….

* Generate a stream on this collection and display the names of cities that has temperature below 10 deg cel.
* Generate a stream on this collection and display the average temperature for the month of August.

## Conclusion

* Intermediate and Terminal operations together when applied on streams, processes the data in the collection.
* Streams never change the data they are operating upon.

# Parallel Sort

## Objective

* Introduction to parallel arrays.
* How will it affect processing of arrays.

## Description

## Parallel Sort

* Java 8 API adds methods to java.util.Arrays to make use of the fork/join framework which has been introduced in Java 7 for sorting arrays.
* The new methods have the same signature as the existing sort() methods, with a parallel prefix.
* All the real work like breaking the array in sub arrays, sorting the sub arrays by distributing the sort algorithm to the available number of CPUs using the fort/join framework and merging the sorted sub arrays to the final result array is done in the background.

|  |  |
| --- | --- |
| * To use the methods, simply use Arrays.parallelSort(myArray) instead of Arrays.sort(myArray) |  |

* Create a class **ParallelSort.java** with following members:

|  |
| --- |
| **public** **class** ParallelSort {  **final** **static** **int** ***ARR\_SIZE*** = 10 \* 1024\*1024; // 10 MB  **final** **static** **int** ***LOOPS*** = 100; //execute 100 times  **public** **static** **void** main(String[] args) {  // create an array with random numbers  **int**[] array = **new** **int**[***ARR\_SIZE***];  Random rand = **new** Random(1);  // make sure to use the same random sequence each time, to  // ensure reproduceability  **for** (**int** i = 0; i < array.length; i++) {  array[i] = rand.nextInt(9999999);  }  // sort the array - note: inline sorting, need to copy  // the array first  Instant st = Instant.*now*();  **for** (**int** i = 0; i < ***LOOPS***; i++) {  **int**[] toSort = Arrays.*copyOf*(array, array.length);  Arrays.*sort*(toSort);  }  Instant end = Instant.*now*();  System.***err***.println("Normal sort: " + Duration.*between*(st,  end));  Instant st1 = Instant.*now*();  **for** (**int** i = 0; i < ***LOOPS***; i++) {  **int**[] toSort = Arrays.*copyOf*(array, array.length);  Arrays.*parallelSort*(toSort);  }  Instant end1 = Instant.*now*();  System.***err***.println("Parallel sort: " + Duration.*between*(st1, end1));  }  } |
|  |

## Conclusion

* As seen in the example, sorting the arrays with parallel processing takes less time than sorting it sequentially.
* The parallelSort() helps developers not to worry about the mechanism behind providing the parallel processing.

# Date/Time API

## Objective

* To explore the new API’s or methods added in existing API.
* What is new in Date/Time API included in java.time package.
* What are repeating annotations and how to use @Repeatable annotation

## Description

## Date/Time API

* Date/Time API designed to overcome the flaws in the legacy date time implementations.
* New package java.time consists of various classes like LocalDate, LocalTime, LocalDateTime, Instant, Perios, Duration etc.
* Lets demonstrate the use of these classes by following demos:
* Example: **LocalDateExample.java**

|  |
| --- |
| /\*\*LocalDate is an immutable class that represents Date with default  \* format of yyyy-MM-dd.We can use now() method to get the current date.  \* We can also provide input arguments for year, month and date to  \* create LocalDate instance.  \* This class provides overloaded method for now() where we can pass  \* ZoneId for getting date in specific time zone.  \* This class provides the same functionality as java.sql.Date  \*/  **public** **class** LocalDateExample {  **public** **static** **void** main(String[] args) {    //Current Date  LocalDate today = LocalDate.*now*();  System.***out***.println("Current Date="+today);    //Creating LocalDate by providing input arguments  LocalDate firstDay\_2014 = LocalDate.*of*(2014, Month.***JANUARY***, 1);  System.***out***.println("Specific Date="+firstDay\_2014);    //Try creating date by providing invalid inputs  //LocalDate feb29\_2014 = LocalDate.of(2014,Month.FEBRUARY,29);  //Exception in thread "main" java.time.DateTimeException:  //Invalid date 'February 29' as '2014' is not a leap year    //Current date in "Asia/Kolkata", you can get it from ZoneId  LocalDate dayKolkata =LocalDate.*now*(ZoneId.*of*("Asia/Kolkata"));  System.***out***.println("Current Date in IST="+todayKolkata);    //java.time.zone.ZoneRulesException: Unknown time-zone ID: IST  //LocalDate todayIST = LocalDate.now(ZoneId.of("IST"));    //Getting date from the base date i.e 01/01/1970  LocalDate dateFromBase = LocalDate.*ofEpochDay*(365);  System.***out***.println("365th day from base date= "+dateFromBase);    LocalDate hundredDay2014 = LocalDate.*ofYearDay*(2014, 100);  System.***out***.println("100th day of 2014="+hundredDay2014);  }  } |
|  |

* Example : **LocalTimeExample.java**

|  |
| --- |
| /\*\* LocalTime is an immutable class whose instance represents  \* a time in the human readable format. It’s default format is  \* hh:mm:ss.zzz.Just like LocalDate, this class provides time  \* zone support and creating instance by passing hour, minute  \* and second as input arguments.  \*/  **public** **class** LocalTimeExample {  **public** **static** **void** main(String[] args) {  //Current Time  LocalTime time = LocalTime.*now*();  System.***out***.println("Current Time="+time);    //Creating LocalTime by providing input arguments  LocalTime specificTime = LocalTime.*of*(12,20,25,40);  System.***out***.println("Specific Time of Day="+specificTime);    //Try creating time by providing invalid inputs  //LocalTime invalidTime = LocalTime.of(25,20);  //Exception in thread "main" java.time.DateTimeException:  //Invalid value for HourOfDay (valid values 0 - 23): 25  //Current date in "Asia/Kolkata", you can get it from ZoneId  LocalTime timeKolkata =  LocalTime.*now*(ZoneId.*of*("Asia/Kolkata"));  System.***out***.println("Current Time in IST="+timeKolkata);    //java.time.zone.ZoneRulesException: Unknown time-zone ID: IST  //LocalTime todayIST = LocalTime.now(ZoneId.of("IST"));  //Getting date from the base date i.e 01/01/1970  LocalTime specificSecondTime = LocalTime.*ofSecondOfDay*(10000);  System.***out***.println("10000th second time="+specificSecondTime);  }  } |
|  |

* Example : **LocalDateTimeExample.java**

|  |
| --- |
| /\*\*LocalDateTime is an immutable date-time object that represents  \* a date-time, with default format as yyyy-MM-dd-HH-mm-ss.zzz.  \* It provides a factory method that takes LocalDate and LocalTime  \* input arguments to create LocalDateTime instance  \*/  **public** **class** LocalDateTimeExample {  **public** **static** **void** main(String[] args) {  //Current Date  LocalDateTime today = LocalDateTime.*now*();  System.***out***.println("Current DateTime="+today);    //Current Date using LocalDate and LocalTime  today = LocalDateTime.*of*(LocalDate.*now*(), LocalTime.*now*());  System.***out***.println("Current DateTime="+today);    //Creating LocalDateTime by providing input arguments  LocalDateTime specificDate = LocalDateTime.*of*(2014,  Month.***JANUARY***, 1, 10, 10, 30);  System.***out***.println("Specific Date="+specificDate);    //Try creating date by providing invalid inputs  //LocalDateTime feb29\_2014 =  //LocalDateTime.of(2014,Month.FEBRUARY, 28, 25,1,1);  //Exception in thread "main" java.time.DateTimeException:  //Invalid value for HourOfDay (valid values 0 - 23): 25  //Current date in "Asia/Kolkata", you can get it from ZoneId  LocalDateTimetodayKolkata=  LocalDateTime.*now*(ZoneId.*of*("Asia/Kolkata"));  System.***out***.println("Current Date in IST="+todayKolkata);    //java.time.zone.ZoneRulesException: Unknown time-zone ID: IST  //LocalDateTime todayIST = LocalDateTime.now(ZoneId.of("IST"));    //Getting date from the base date i.e 01/01/1970  LocalDateTime dateFromBase = LocalDateTime.*ofEpochSecond*(10000,  0, ZoneOffset.***UTC***);  System.***out***.println("10000th second time from 01/01/1970 =  "+dateFromBase);  }  } |

* Example : **InstantExample.java**

|  |
| --- |
| /\*\*Instant class is used to work with machine readable time format,  \* it stores date time in unix timestamp  \*/  **public** **class** InstantExample {  **public** **static** **void** main(String[] args) **throws**  InterruptedException  {  //Current timestamp  Instant timestamp = Instant.*now*();  System.***out***.println("Current Timestamp = "+timestamp);  Thread.*sleep*(1500);  //Instant from timestamp  InstantspecificTime=Instant.*ofEpochMilli*(  timestamp.toEpochMilli());  System.***out***.println("Specific Time = "+specificTime);    //Duration example  Duration thirtyDay = Duration.*ofDays*(20);  System.***out***.println(thirtyDay);  }  } |

* Example: **DateAPIUtilities.java**

|  |
| --- |
| /\*\* Most of the Date Time principle classes provide various utility  \* methods such as plus/minus days, weeks, months etc.There are  \* some other utility methods for adjusting the date using  \* TemporalAdjuster and to calculate the period between two dates.  \*/  **public** **class** DateAPIUtilities {  **public** **static** **void** main(String[] args) {  LocalDate today = LocalDate.*now*();    //Get the Year, check if it's leap year  System.***out***.println("Year "+today.getYear()+" is Leap Year?  "+today.isLeapYear());    //Compare two LocalDate for before and after  System.***out***.println("Today is before 01/01/2015?  "+today.isBefore(LocalDate.*of*(2015,1,1)));    //Create LocalDateTime from LocalDate  System.***out***.println("Current  Time="+today.atTime(LocalTime.*now*()));    //plus and minus operations  System.***out***.println("10 days after today will be  "+today.plusDays(10));  System.***out***.println("3 weeks after today will be  "+today.plusWeeks(3));  System.***out***.println("20 months after today will be  "+today.plusMonths(20));    System.***out***.println("10 days before today will be  "+today.minusDays(10));  System.***out***.println("3 weeks before today will be  "+today.minusWeeks(3));  System.***out***.println("20 months before today will be  "+today.minusMonths(20));    //Temporal adjusters for adjusting the dates  System.***out***.println("First date of this month=  "+today.with(TemporalAdjusters.*firstDayOfMonth*()));  LocalDate lastDayOfYear =  today.with(TemporalAdjusters.*lastDayOfYear*());  System.***out***.println("Last date of this year= "+lastDayOfYear);  Period period = today.until(lastDayOfYear);  System.***out***.println("Period Format= "+period);  System.***out***.println("Months remaining in the year=  "+period.getMonths());  }  } |

# Day 3: Optional class

## Objective

* What is Optional class
* How to define and use them and the various methods defined.

## Description

* Optional class from java.util package is supposed to cure NullPointerExceptions.
* Optional can be viewed as a single-value container or a wrapper that either contains a value or doesn't (it is then said to be "empty")
* Optionals should be used as return type of methods that might not return a value.
* Optional object can be created in various ways depending on if the object can accept a null value or not.
* Lets demonstrate this through an example.
* Example:
* Create a class **Car.java as follows:**

|  |
| --- |
| **public** **class** Car {  String name;  **public** Car() {  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Car(String name) {  **super**();  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Car [name=" + name + "]";  }  } |

* Create a class **CarDemo.java** as follows:

|  |
| --- |
| **public** **class** CarDemo {  **public** **static** Optional<Car> createOptional(**boolean** flag)  {  **if**(flag)  //Object passed to of() cannot be null  //Else NullPointerException is thrown  **return** Optional.*of*(**new** Car("Maruti"));  //empty optional whose value is null or not initialised  **return** Optional.*empty*();  }  } |

* To invoke the createOptional() method and test it, write the main method in **CarDemo.java** as follows:

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Optional<Car> carempty = *createOptional*(**false**);  System.***out***.println(carempty);//Optional.empty  Optional<Car> carobj= *createOptional*(**true**);  System.***out***.println(carobj);//Optional[Car [name=Maruti]]    //ofNullable() offers the possibility of null values  Optional<Car> carnull = Optional.*ofNullable*(**null**);  System.***out***.println(carnull);//Optional.empty  } |

* In order to get the value of an Optional get() is used as follows:

|  |
| --- |
| Optional<Car> carobj= *createOptional*(**true**);  System.***out***.println(carobj);//Optional[Car [name=Maruti]] |

* If an object is not initialsed the default value can be retrieved using orElse method

|  |
| --- |
| Car defcar = **new** Car("Tata");  Car cardefault = *createOptional*(**false**).orElse(defcar);  System.***out***.println(cardefault);//Car [name=Tata] |

* To check whether Optional holds any value or not use isPresent() or ifPresent() methods

|  |
| --- |
| Optional<Car> carpresent= Optional.*ofNullable*(**null**);  **if**(carpresent.isPresent())  System.***out***.println(carobj.toString());//NO OUTPUT  //OR  carpresent.ifPresent(System.***out***::println); //NO OUTPUT |

## Assignment

* In Eclipse under Java Perspective create a project **Java8\_04** .
* Under **com.reference** package create a class **Fruit.java** as follows:

|  |
| --- |
| **public** **class** Fruit {  String name;  **public** Fruit() {  // **TODO** Auto-generated constructor stub  }  **public** Fruit(String name) {  **super**();  **this**.name = name;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Fruit [name=" + name + "]";  }  } |

## TO\_DO…….

* Under **com.test** package create a class **TestFruit.java**
* Create a static method createFruit() responsible of creating Fruit objects and returning the Optional<Fruit>if present or else return empty Optional
* Test the method createFruit() if it returns Optional object or empty Optional using isPresent() and ifPresent() methods
* If Fruit object is present display the name of this fruit or else pass a default Fruit object and display the name of default fruit.

## Conclusion

* Optional class works as a wrapper around references to avoid NullPointerException.
* We have seen various ways of ceating Optional objects and the methods available in Optional class