Synechron®
Synergize . Synchronize . Surpass .

JAVA 8 - SYLLABUS

1 Java 8

1.1 Functional Interfaces

An interface with exactly one abstract method becomes functional interface. We don't need to define @FunctionalInterface annotation to mark interfaces as Functional Interfaces. @FunctionalInterface annotation is mark to avoid accidental addition of abstract methods in the functional interfaces. You can think of it like @override annotation as best practice to use it.

From Java 8 onwards Java.lang.Runnable interface with only one abstract method run() is become a functional interface.

```
@FunctionalInterface
public interface WorkerInterface {

    public void doSomeTask();

 }
```

If you try to add one more abstract method in Functional Interface, it throws compile time error.

```
@FunctionalInterface
public interface WorkerInterface {

    public void doSomeTask();
    public   void   doSomeMoreTask();//
   compile time error
 }
```

Error:

```
 Unexpected @FunctionalInterface
annotation
 @FunctionalInterface
WorkerInterface    is    not    a
functional  interface.  Multiple
abstract   methods   found   in
interface WorkerInterface
```

Benefit of functional interface:

1)One of the major benefits of functional interface is used in lambda expressions to instantiate them.
2)Java 8 also declared number of Functional Interfaces that can be used by Lambda expressions.

1.2 Default and static method in interface

Prior to Java 8 if we wanted to add new methods in the interfaces, it would require change in the all implementing classes. Prior to Java 8 interfaces were allowed to only declare methods but starting Java 8 we can also define methods in interfaces. Java 8 has introduced the concept of default method in interface which allows us to add new methods in interfaces without having the existing classes to override those methods.

We know that Java doesn’t provide multiple inheritances in Classes because it leads to diamond Problem.

Interfaces are now similar to abstract classes, how multiple inheritance would be handled in interface?

The solution is that we have to provide implementation logic in the implementation class else compiler will throw exception.

```
// Openable.java
@FunctionalInterface
public interface Openable {
    void open(String str);
    default void log(String str){
        System.out.println("Openable
logging"+str);
    }


    static void print(String str){
        System.out.println("Printing value of
str "+str);
    }



 //If we try to override Object class method
then it gives compile time error like "A
default method cannot override a method from
java.lang.Object"
//  default String toString(){
//      return "i1";
//  }


}
```

```
// Closeable.java
@FunctionalInterface
public interface Closeable {


    void close();


    default void log(String str){
        System.out.println("Closeable
logging"+str);
    }


}
```

```
// MyClass.java
public class MyDocument implements Openable,
Closeable {
  @Override
        public void open(String str) {
        }


    @Override
    public void close() {
    }
     //MyDocument will not compile without
having it's own log() implementation
    @Override
    public void log(String str){
        System.out.println("MyDocument
logging::"+str);
Openable.print();
    }


}
```

As you can see that Openable interface has static method implementation that is used

in MyDocument.log() method implementation.

Java 8 uses default and static methods heavily in Collection API .

If any class in the hierarchy has a same method with same signature, then default methods become irrelevant.

Since any class implementing an interface already has Object as super class, if we have equals(), hashCode() default methods in interface, it will become irrelevant. That's why for better clarity, interfaces are not allowed to have Object class default methods.

1.3 Lambda Expression

Consider using lambda expression where a method is being used only once and the method definition is short. **A lambda expression is an anonymous function**. Simply consider like a method without a declaration, i.e., access modifier, return value declaration, and name. It's shorthand notation that allows you to write a method in the same place you are going to use it. It saves you the effort of writing a separate method to the containing class.

A lambda expression in Java is usual written using syntax

(argument) -> {body}

.

Following are some examples of Lambda expressions.

```
(int a, int b) -> { return a + b; }

() -> System.out.println("Hello World using Lambda expression");

(String s) -> { System.out.println(s); }

() -> 42

() -> { return 3.1415 };
```

Let's check the structure of lambda expressions as below:

1. A lambda expression can have zero, one or more parameters.
2. Type of the parameters can be explicitly declared or it can be conclude from the context. e.g. (int a) is same as just (a)
3. Parameters are enclosed in parentheses and separated by commas.
   e.g. (a, b) or (int a, int b) or (String a, int b, float c)
4. Empty parentheses are used to define an empty set of parameters.
   e.g. () -> 42
5. When there is a single parameter, if its type is inferred then it is not mandatory to use parentheses. e.g. a -> return a*a
6. The body of the lambda expressions can contain zero, one or more statements.
7. If body of lambda expression has single statement then curly brackets are not mandatory and return type of the anonymous function is the same as that of the body expression.
8. If there is more than one statement in body then it must be enclosed in curly brackets and the return type of the anonymous function is consider same as the type of the value returned within the code block, or void if nothing is returned.

Once the Functional interface is defined, we can use it in our API and take advantage of Lambda expressions.

Code Example 1:

```
//functional interface
@FunctionalInterface
public interface WorkerInterface {
     public void doSomeTask();
 //    public void doSomeMoreWork();
}
```

```
public class WorkerInterfaceImpl implements
    WorkerInterface {

    @Override
    public void doSomeTask() {
     System.out.println("Worker invoked using
    implementation class");
    }
}
```

```
public class WorkerInterfaceTest {

    public            static            void
    execute(WorkerInterface worker)
    {
         worker.doSomeTask();
     }

    public static void main(String
    [] args) {
         //invoke doSomeTask using
    Implementation class
         execute(new
    WorkerInterfaceImpl());
```

```
        //invoke  doSomeTask  using
    Annonymous class
        execute(new
    WorkerInterface() {
            @Override
            public                void
    doSomeTask() {

    System.out.println("Worker
    invoked using Anonymous class");
            }
        });

        //invoke  doSomeTask  using
    Lambda expression
        execute(         ()        ->
    {System.out.println("Worker
    invoked         using        Lambda
    expression");});
    }
}
```

Output:

```
Worker  invoked  using  implementation
    class
Worker invoked using Anonymous class
Worker     invoked     using     Lambda
    expression
```

Code Example 2:

Using Lambda expression thread can be initialized like following:

```
public class Test2 {

    public static void main(String[]
    args) {
     //  TODO  Auto-generated  method
    stub

      //Old way to create thread:
         new Thread(new Runnable() {
             @Override
             public void run() {

    System.out.println("Hello   from
    Thread1");
             }
         }).start();



      //New  way  to  create  thread
    using lambda expression:

         new Thread(
           ()                      ->
    System.out.println("Hello   from
    Thread2")
         ).start();

         }
}
```

Output:

```
Hello from Thread1
Hello from Thread2
```

Code Example 3

In this example we use forEach () from List & double colon (::) operator

Following code is to print all elements of given array. In below example we use the way of creating lambda expression using arrow syntax and also we have used a brand new double colon (::) operator in Java 8 has to convert a normal method into lambda expression.

```
public class Test3 {

   public static void main(String[]
   args) {

    List<Integer>      mylist      =
   Arrays.asList(1, 2, 3, 4, 5, 6,
   7);

    System.out.println("Using     Old
   way");
    //Old way:
    for(Integer n: mylist) {
        System.out.print(" "+n);
    }

    System.out.println("\n"+"Using
   New way");
    //New way:
       mylist.forEach(n             ->
    System.out.print(" "+n));
    System.out.println("\n"+"Using ::
   double colon operator");
    //or we can use :: double colon
   operator in Java 8

     mylist.forEach(System.out::print
   );
   }
}
```

Output:

```
Using Old way
 1 2 3 4 5 6 7
Using New way
 1 2 3 4 5 6 7
Using :: double colon operator
1234567
```

1.4 forEach() Method in Iterable Interface

Java 8 has introduced forEach() method in java.lang.Iterable interface so that while
writing code we have to focus on business logic only.

forEach method implementation in Iterable interface is

default void forEach(Consumer<? super T> action)

{

```
        Objects.requireNonNull(action);
       for (T t : this) {
        action.accept(t);
      }
 }
```

forEach method takes java.util.function.Consumer object as an argument, so it helps in having our business logic at a separate location that we can reuse. We can also use lambda expression for printing values while iterating over collection using forEach.

```
Map<String, Integer> items = new HashMap<
String, Integer >();


import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.function.Consumer;
import java.lang.Integer;

public class Java8ForEachExample {

    public static void main(String[] args) {

        //creating sample Collection
        List<Integer>     myList     =     new
   ArrayList<Integer>();

        for(int i=0; i<10; i++)
           myList.add(i);

        //traversing using Iterator
        Iterator<Integer>         it         =
   myList.iterator();
        while(it.hasNext()){
            Integer i = it.next();
            System.out.println("Iterator
   Value::"+i);
            if(i==6)
            {
               System.out.println("        some
   processing ");
            }

        }

        //traversing through forEach method of
   Iterable with anonymous class
        myList.forEach(new  Consumer<Integer>()
   {

            public void accept(Integer t) {
                System.out.println("forEach
   anonymous class Value::"+t);

            }

        });

        //traversing  with  Consumer  interface
   implementation
        MyConsumer action = new MyConsumer();
        myList.forEach(action);

    }

}
```

```
//Consumer implementation that can be reused
class MyConsumer implements Consumer<Integer>{

    public void accept(Integer t) {
        System.out.println("Consumer impl
Value::"+t);
        if(t==6)
        {
            System.out.println(" some
processing ");
        }
    }
```

Method References

- Used to refer method of functional interface.
- Compact and easy form of lambda expression.
- when using lambda expression to just referring a method, replace lambda expression with method reference.xxxxxxx

Types of Method Reference

- Reference to a static method.
- Reference to an instance method.
- Reference to a constructor.

Type 1: Reference to a static method

- **Lambda Syntax:**
  - (arguments)-> <ClassName>.<staticMethodName>(arguments);
- **Equivalent Method Reference:**
  - <ClassName> :: <staticMethodName>

Type 2: Reference to an instance method

- **Lambda Syntax:**
  - (arguments) -> <expression>.<instanceMethodName>(arguments)
- **Equivalent Method Reference:**
  - <expression> :: <instanceMethodName>

Type 3:Reference to a Constructor

- **Syntax of Constructor References:**

<ClassName>::new

Java 8 Stream

- The Stream API in Java 8 lets you write code that's
  - **Declarative** : More concise and readable
  - **Composable** : Greater Flexibility
  - **Parallelizable :** Better Performance

What is Stream ?

A **sequence of elements** from a **source** that supports **data processing operations**.

- ***Sequence of elements***— Like a collection, a stream provides an interface to a sequenced set of values of a specific element type.(Collections are about data, where streams are about computations)
- ***Source***— Streams consume from a data-providing source such as collections, arrays, or I/O resources.
- ***Data processing operations***— Streams support database-like operations and common operations from functional programming languages to manipulate data, such as filter, map, reduce, find, match, sort, and so on. Stream operations can be executed either sequentially or in parallel.

Stream operations have two important characteristics:

- ***Pipelining***— Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline. A pipeline of operations can be viewed as a database-like query on the data source.

- ***Internal iteration—*** In contrast to collections, which are iterated explicitly using an iterator, stream operations do the iteration behind the scenes for you.

Example : In Java 7 (Without stream)

Example : In Java 8 (With stream)

Stream Operations

Intermediate Operation

- Allows the operations to be connected to form a query.
- Don’t perform any processing until a terminal operation is invoked.

Terminal Operation

- produce a result from a stream pipeline.
- A result is any nonstream value such as a List, an Integer, or even void.

Working with Streams?

- A *data source* (such as a collection) to perform a query on
- A chain of *intermediate operations* that form a stream pipeline
- A *terminal operation* that executes the stream pipeline and produces a result

List of Intermediate Operations

filter

map

limit

sorted

distinct

List of Terminal operation

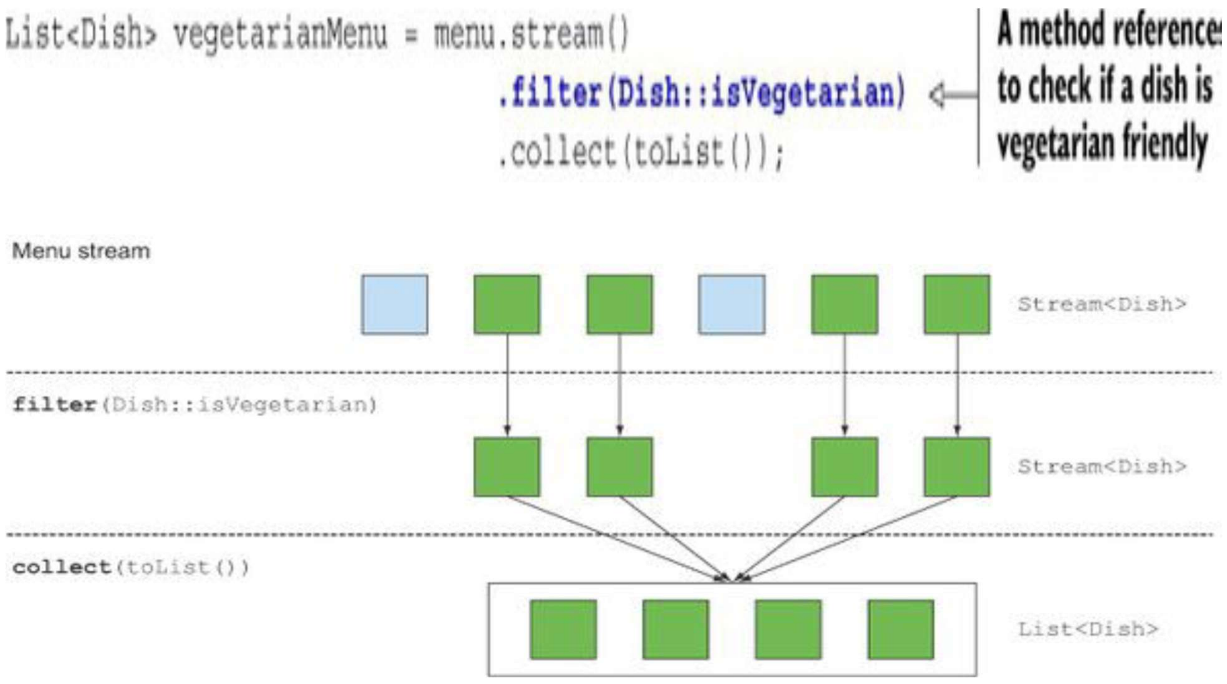
forEach

count

collect

Filtering:

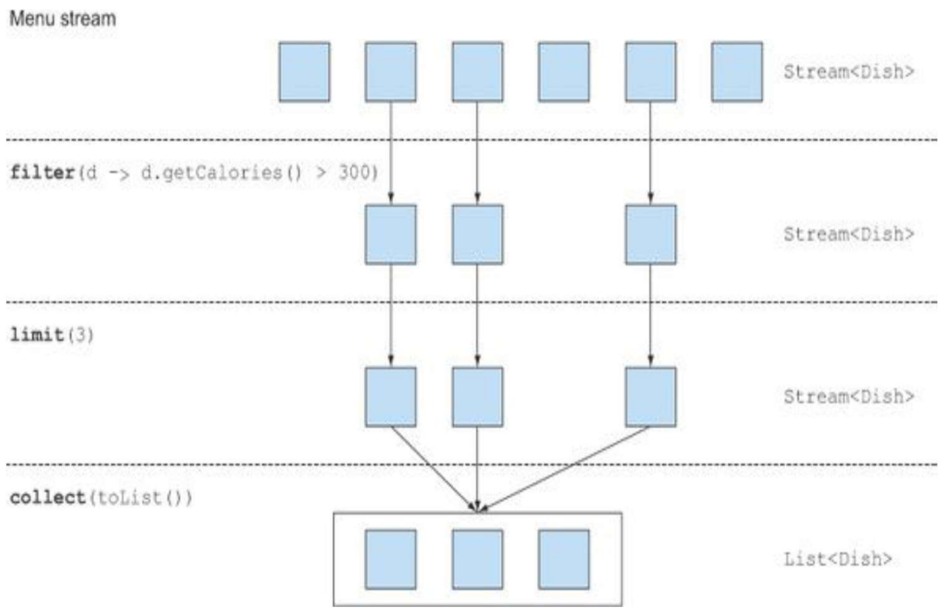
Filtering a stream with predicate



```
List<Dish> vegetarianMenu = menu.stream()
                                .filter(Dish::isVegetarian)
                                .collect(toList());
```

Truncating a stream (limit)

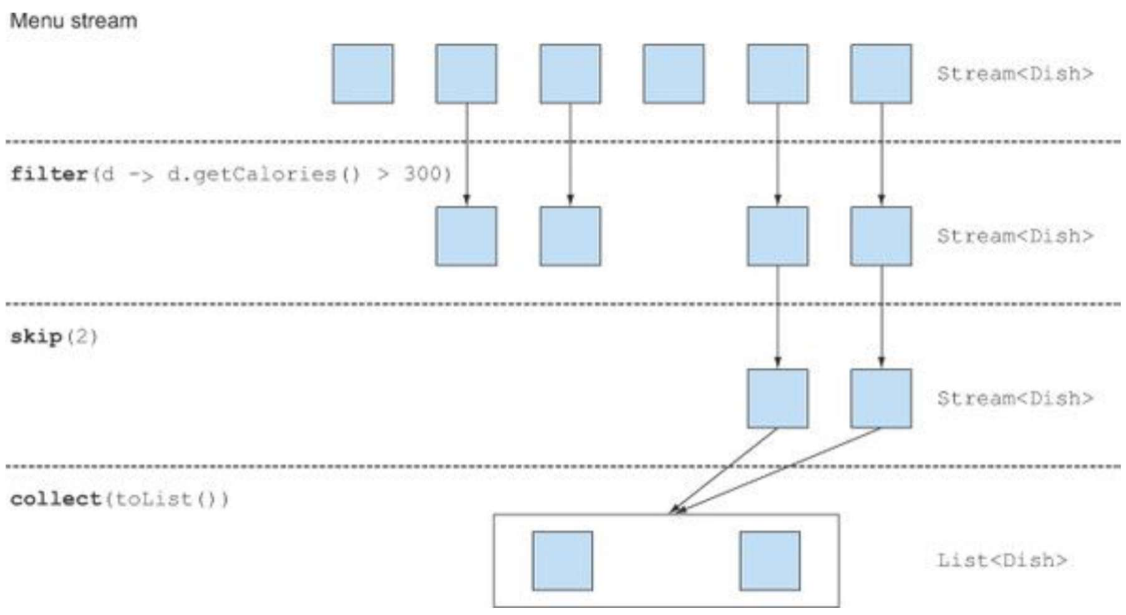
▶ Streams support the limit(n) method, which returns another stream that's no longer than a given size. List<Dish> dishes = menu.stream() .filter(d -> d.getCalories() > 300) ***.limit(3)*** .collect(toList());



Skipping elements

- Streams support the skip(n) method to return a stream that discards the first n elements.
- If the stream has fewer elements than n, then an empty stream is returned.

```
List<Dish> dishes = menu.stream()
    .filter(d -> d.getCalories() > 300)
    .skip(2)
    .collect(toList());
```



Finding And Matching

- The Streams API provides finding and matching through the allMatch, anyMatch, noneMatch, findFirst, and findAny methods of a stream.
- **Checking to see if a predicate matches at least one element:**

```
if(menu.stream().anyMatch(Dish::isVegetarian)){
        System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

Note :The anyMatch method returns a boolean and is therefore a terminal operation.

- **Checking to see if a predicate matches all elements**

```
boolean isHealthy = menu.stream()
.allMatch(d -> d.getCalories() < 1000);
```

- **Note:** The allMatch method works similarly to anyMatch but will check to see if all the elements of the stream match the given predicate.

- **noneMatch**
  - ensures that no elements in the stream match thegiven predicate.

    ```
    boolean isHealthy = menu.stream()
    .noneMatch(d -> d.getCalories() >= 1000);
    ```
  - **Finding an element**
  - The findAny method returns an arbitrary element of the current stream. It can be used in conjunction with other stream operations.

```
Optional<Dish> dish =menu.stream()
.filter(Dish::isVegetarian)
.findAny();
```

- **Note**: The Optional<T> class (java.util.Optional) is a container class to represent the existence or absence of a value. In the previous code, it's possible that findAny doesn't find any element. Instead of returning null, which is well known for being error prone, the Java 8 library designers introduced Optional<T>.

Reducing

- combine all elements of a stream iteratively to produce a result using the reduce method, for example, to calculate the sum or find the maximum of a stream.

- **Summing the elements**

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}

int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

- //Multiplying using reduce

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

Parallel Stream:

- Internal iteration allows you to process a stream in parallel without the need to explicitly use and coordinate different threads in your code.
- Even if processing a stream in parallel is so easy, there’s no guarantee that doing so will make your programs run faster under all circumstances.
- Parallel execution of an operation on a set of data, as done by a parallel stream, can provide a performance boost, especially when the number of elements to be processed is huge or the processing of each single element is particularly time consuming.

- From a performance point of view, using the right data structure, for instance, employing primitive streams instead of nonspecialized ones whenever possible, is almost always more important than trying to parallelize some operations.
- The fork/join framework lets you recursively split a parallelizable task into smaller tasks, execute them on different threads, and then combine the results of each subtask in order to produce the overall result.

- parallelStream() method is used to create a parallel stream of elements.

Code Example

```
public class StreamExample {
    public static void main(String[] args) {
        List<Integer> myList = new
ArrayList<>();
        for(int i=0; i<100; i++) myList.add(i);

         //sequential stream
         Stream<Integer> sequentialStream =
myList.stream();
         //parallel stream
         Stream<Integer> parallelStream =
myList.parallelStream();
         //using lambda with Stream API, filter
example
         Stream<Integer> highNums =
parallelStream.filter(p -> p > 90);
         //using lambda in forEach
        highNums.forEach(p ->
System.out.println("High Nums parallel="+p));
        //using lambda with Stream API, filter
example
        Stream<Integer> highNumsSeq =
sequentialStream.filter(p -> p > 90);
        //using lambda in forEach
        highNumsSeq.forEach(p ->
System.out.println("High Nums
        sequential="+p));
     }
}
```

Output

```
High Nums parallel=91
High Nums parallel=93
High Nums parallel=92
High Nums parallel=94
High Nums parallel=95
High Nums parallel=96
High Nums parallel=97
High Nums parallel=98
High Nums parallel=99
High Nums sequential=91
High Nums sequential=92
High Nums sequential=93
High Nums sequential=94
High Nums sequential=95
High Nums sequential=96
High Nums sequential=97
High Nums sequential=98
High Nums sequential=99
```

Notice that parallel processing values are not in order, so parallel processing will be very helpful while working with huge collections.

- **Java Date Time API**

Why do we need new Java Date Time API?

Before we start looking at the Java 8 Date Time API, let's see why we need a new API for this. There have been several problems with the existing date and time related classes in java, some of them are:

1. Java Date Time classes are not defined consistently; we have Date Class in both java.util as well as java.sql packages. Again formatting and parsing classes are defined in java.text package.
2. java.util.Date contains both date and time, whereas java.sql.Date contains only date. Having this in java.sql package doesn't make sense. Also both the classes have same name, that is a very bad design itself.
3. There are no clearly defined classes for time, timestamp, formatting and parsing. We have java.text.DateFormat abstract class for parsing and formatting need. Usually SimpleDateFormat class is used for parsing and formatting.
4. All the Date classes are mutable, so they are **not thread safe**. It's one of the biggest problem with Java Date and Calendar classes.
5. Date class doesn't provide internationalization, there is no timezone support. So java.util.Calendar and java.util.TimeZone classes were introduced, but they also have all the problems listed above.

Design principles of new Date Time API are:

1. **Immutability**: All the classes in the new Date Time API are immutable and good for multithreaded environments.
2. **Separation of Concerns**: The new API separates clearly between human readable date time and machine time (unix timestamp). It defines separate classes for Date, Time, DateTime, Timestamp, Timezone etc.
3. **Clarity**: The methods are clearly defined and perform the same action in all the classes. For example, to get the current instance we have now() method. There are format() and parse() methods defined in all these classes rather than having a separate class for them. All the classes use Factory Pattern and Strategy Pattern for better handling. Once you have used the methods in one of the class, working with other classes won't be hard.
4. **Utility operations**: All the new Date Time API classes comes with methods to perform common tasks, such as plus, minus, format, parsing, getting separate part in date/time etc.
5. **Extendable**: The new Date Time API works on ISO-8601 calendar system but we can use it with other non ISO calendars as well.

Java Date Time API Packages

Java Date Time API consists of following packages.

1. **java.time Package**: This is the base package of new Java Date Time API. All the major base classes are part of this package, such as LocalDate, LocalTime, LocalDateTime, Instant, Period, Duration etc . All of these classes are immutable and thread safe. Most of the

times, these classes will be sufficient for handling common requirements.

2. **java.time.chrono Package**: This package defines generic APIs for non ISO calendar systems. We can extend AbstractChronology class to create our own calendar system.
3. **java.time.format Package**: This package contains classes used for formatting and parsing date time objects. Most of the times, we would not be directly using them because principle classes in java.time package provide formatting and parsing methods.
4. **java.time.temporal Package**: This package contains temporal objects and we can use it for find out specific date or time related to date/time object. For example, we can use these to find out the first or last day of the month. You can identify these methods easily because they always have format “withXXX”.
5. **java.time.zone Package**: This package contains classes for supporting different time zones and their rules.

Code Example 1

java.time.LocalDate: 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.

```java
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 javadoc
        LocalDate todayKolkata = 
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);
    }

}
```

Output

```
Current Date=2015-04-17
Specific Date=2014-01-01
Current Date in IST=2015-04-17
365th day from base date= 1971-01-01
100th day of 2014=2014-04-10
```

Code Example 2

java.time.LocalTime: 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 javadoc
        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);

    }
}
```

Output

```
Current Time=17:06:25.047
Specific Time of Day=12:20:25.000000040
Current Time in IST=17:06:25.048
10000th second time= 02:46:40
```

Code Example 3

java.time.LocalDateTime: 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 javadoc
```

```
        LocalDateTime todayKolkata =
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);

    }
}
```

Output

```
Current DateTime=2015-04-17T17:14:44.754
Current DateTime=2015-04-17T17:14:44.755
Specific Date=2014-01-01T10:10:30
Current Date in IST=2015-04-17T17:14:44.756
10000th second time from 01/01/1970= 1970-01-
01T02:46:40
```

Code Example 4

java.time.Period

There are some other utility methods for adjusting the date using TemporalAdjuster and to calculate the period between two dates.

```
public class PeriodExample {

    public static void main(String[] args) {

        LocalDate today = LocalDate.now();
        System.out.println("Today's Date="+today);

        //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());
    }
}
```

Output

```
Today's Date= 2015-04-17
First date of this month= 2015-04-01
```

```
Last date of this year= 2015-12-31
Period Format= P8M14D
Months remaining in the year= 8
```

Code Example 6

java.time.Duration

A Duration is similar to a period but its precision is based on hours, minutes, seconds, miliseconds…It is also a distance on the timeline. A Duration can be created using an amount of seconds (or minutes, hours…) or by specifying an start and an end times:

```
public class DurationsExample {

    public static void main(String args[])
    {
        Duration duration = Duration.ofSeconds( 59
);
        System.out.println(duration);
        duration = Duration.between(
LocalTime.now(), LocalTime.MIDNIGHT );
        System.out.println(duration);
        duration = Duration.between( LocalTime.now(
ZoneId.of(
             ZoneId.SHORT_IDS.get( "AGT" ) ) ),
LocalTime.MIDNIGHT );
        System.out.println(duration);
    }
}
```

Output

```
PT59S
PT-19H-41M-49.498S
PT-11H-11M-49.498S
```

Code Example 7

Date API Utilities: Most of the Date Time principle classes provide various utility methods such as plus/minus days, weeks, months etc.

```
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));
         }
}
```

Output

```
Year 2015 is Leap Year? False
Today is before 01/01/2015? false
Current Time=2015-04-17T19:47:58.159
10 days after today will be 2015-04-27
3 weeks after today will be 2015-05-08
20 months after today will be 2016-12-17
10 days before today will be 2015-04-07
3 weeks before today will be 2015-03-27
20 months before today will be 2013-08-17
```

Code Example 8

java.time.format.DateTimeFormatter :

Parsing and Formatting : It's very common to format date into different formats and then parse a String to get the Date Time objects.

```
public class DateParseFormatExample {

    public static void main(String[] args) {
        //Format examples
        LocalDate date = LocalDate.now();
        //default format
        System.out.println("Default format of
LocalDate="+date);
        //specific format


System.out.println(date.format(DateTimeFormatter.of
Pattern("d::MMM::uuuu")));

System.out.println(date.format(DateTimeFormatter.BA
SIC_ISO_DATE));


        LocalDateTime dateTime =
LocalDateTime.now();
        //default format
        System.out.println("Default format of
LocalDateTime="+dateTime);
        //specific format


System.out.println(dateTime.format(DateTimeFormatte
r.ofPattern("d::MMM::uuuu
      HH::mm::ss")));
System.out.println(dateTime.format(DateTimeFormatte
r.BASIC_ISO_DATE));

        Instant timestamp = Instant.now();
        //default format
        System.out.println("Default format of
Instant="+timestamp);

        //Parse examples
        LocalDateTime dt =
LocalDateTime.parse("27::Apr::2014 21::39::48",

DateTimeFormatter.ofPattern("d::MMM::uuuu
HH::mm::ss"));
        System.out.println("Default format after
parsing = "+dt);
    }
}
```

Output

```
Default format of LocalDate=2015-04-17
17::Apr::2015
20150417
Default format of LocalDateTime=2015-04-
17T19:52:33.854
17::Apr::2015 19::52::33
20150417
Default format of Instant=2015-04-17T14:22:33.855Z
Default format after parsing = 2014-04-27T21:39:48
```

Code Example 9

java.time.temporal. TemporalAdjusters :

Adjusters are classes and interfaces with methods that “adjust” any kind of temporal value preserving its state, i.e. the state and values of the used temporal value does not change after applying the adjuster operations.

```
public class TemporalAdjustersExample {

    public static void main(String args[])
    {
        LocalDate now = LocalDate.now();
        LocalDate adjusted = now.with(
TemporalAdjusters.lastDayOfMonth() );
          System.out.println( "now with last day of
month " + adjusted );
          System.out.println( "now " + now );
    }
}
```

Output

```
now with last day of month 2015-04-30
now 2015-04-17
```

Code Example 10

Legacy Date Time Support: Legacy Date/Time classes are used in almost all the applications, so having backward compatibility is a must. That’s why there are several utility methods through which we can convert Legacy classes to new classes and vice versa.

```
public class DateAPILegacySupport {

    public static void main(String[] args) {
        //Date to Instant
        Instant timestamp = new Date().toInstant();
        //Now we can convert Instant to
LocalDateTime or other similar classes
        LocalDateTime date =
LocalDateTime.ofInstant(timestamp,

ZoneId.of(ZoneId.SHORT_IDS.get("PST")));
        System.out.println("Date = "+date);

        //Calendar to Instant
        Instant time =
Calendar.getInstance().toInstant();
        System.out.println(time);

        //TimeZone to ZoneId
        ZoneId defaultZone =
TimeZone.getDefault().toZoneId();
        System.out.println(defaultZone);

        //ZonedDateTime from specific Calendar
        ZonedDateTime gregorianCalendarDateTime =
new
        GregorianCalendar().toZonedDateTime();

System.out.println(gregorianCalendarDateTime);

        //Date API to Legacy classes
        Date dt = Date.from(Instant.now());
        System.out.println(dt);

        TimeZone tz =
TimeZone.getTimeZone(defaultZone);
        System.out.println(tz);

        GregorianCalendar gc =
GregorianCalendar.from(gregorianCalendarDateTime);
        System.out.println(gc);
    }
}
```

Output

```
Date = 2015-04-17T07:30:05.776
2015-04-17T14:30:05.8
```

```
Asia/Calcutta
2015-04-17T20:00:05.916+05:30[Asia/Calcutta]
Fri Apr 17 20:00:05 IST 2015
sun.util.calendar.ZoneInfo[id="Asia/Calcutta",offse
t=19800000,dstSavings=0,useDaylight=false,transitio
ns=6,lastRule=null]
java.util.GregorianCalendar[time=1429281005916,areF
ieldsSet=true,areAllFieldsSet=true,lenient=true,zon
e=sun.util.calendar.ZoneInfo[id="Asia/Calcutta",off
set=19800000,dstSavings=0,useDaylight=false,transit
ions=6,lastRule=null],firstDayOfWeek=2,minimalDaysI
nFirstWeek=4,ERA=1,YEAR=2015,MONTH=3,WEEK_OF_YEAR=1
6,WEEK_OF_MONTH=3,DAY_OF_MONTH=17,DAY_OF_YEAR=107,D
AY_OF_WEEK=6,DAY_OF_WEEK_IN_MONTH=3,AM_PM=1,HOUR=8,
HOUR_OF_DAY=20,MINUTE=0,SECOND=5,MILLISECOND=916,ZO
NE_OFFSET=19800000,DST_OFFSET=0]
```

- **Collection API improvement**

We have already seen forEach() method and Stream API for collections.

Some new methods added in Collection API are:

1) Method : void forEach(Consumer<? super T> action)

class/interface: Iterable

Example

```
//Iterates over each element of the List and calls the lambda
expression specified by 'action'.

List<Double> temperature =
  new ArrayList<Double<(Arrays.asList(new
Double[] { 20.0, 22.0, 22.5 }));
temperature.forEach(s ->
System.out.println(s));
```

2)Method : boolean removeIf(Predicate<? super E> filter)

class/interface: Collection

Example

```
Iterates through the Collection and removes the element that
matches the filter.
temperature.removeIf(s -> s > 22);
// remove elements that are > 22
```

3)Method : void replaceAll(UnaryOperator<E> operator)

class/interface: List

Example

```
This is a very useful method. It replaces all elements in the List with
the result of applying the operator (apply method)
temperature.replaceAll(s->Math.pow(s, 0.5));
// replaces all elements by its square root
```

4)Method : void sort(Comparator<? super E> c)

class/interface: List

Example

Sorts the element using the provided comparator. This example sorts the elements in descending order

```
temperature.sort((a, b) -> a > b ? -1 : 1);
```

5)Method : void forEach(BiConsumer<? super K, ? super V> action)

class/interface: Map

Example

This method performs the operation specified in the 'action' on each Map Entry (key and
value pair). It iterates in the order of the key set.

```
authorBooks.forEach((a, b) -> System.out.println(a + " wrote " + b + "
books"));
Map<String , Integer> authorBooks = new HashMap<String ,
Integer>();
authorBooks.put("Robert Ludlum", 27);
authorBooks.put("Clive Cussler", 50);
authorBooks.put("Tom Clancy", 17);
```

6)Method : V compute(K key, BiFunction<? super K, ? super V, ? extends V>

remappingFunction)

class/interface: Map

Example

This method replaces the value of a key by the value computer from the remappingFunction. In this example we replace the number of books written by Clive Cussler to original count(50) +1

```
authorBooks.compute("Clive Cussler", (a, b) -> b + 1);
```

If the compute function returns null then the entry for that key is removed from the map. If the key is not present then a new entry is added.

7)Method : V computeIfAbsent(K key, Function<? super K, ? extends V>

mappingFunction)

class/interface: Map

Example

This method adds an entry in the Map. the key is specified in the function and the value is the result of the application of the mapping function. In our slightly weird example, we add the number of books written by 'Agatha Christie' as the number of alphabets in the authors name

```
authorBooks.computeIfAbsent("Agatha Christie", b -> b.length());
```

The entry is added only if the computed value is not null.

8)Method : V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V>

remappingFunction)

class/interface: Map

Example

This function is simlar to the compute function, however the difference is that the compute function adds or modifies an existing entry whereas this function does nothing if an entry with that key is not present. Note that this function also removes an element if the new value computed from the passed lambda expression is null

```
authorBooks.computeIfPresent("Tom Clancy", (a, b) -> b + 1);
```

9)Method : V getOrDefault(Object key, V defaultValue)

class/interface: Map

Example

Returns the value mapped to the key, or if the key is not present, returns the default value.

```
authorBooks.getOrDefault("AuthorA", 0)
```

the map does not contain 'AuthorA' so this returns 0.

10)Method : V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V>

remappingFunction)

class/interface: Map

Example

If the key is not present or if the value for the key is null, then adds the key-value pair to the map. If the key is present then replaces the value with the value from the remapping function. If the remapping function return null then the key is removed from the map.

```
authorBooks.merge("AuthorB", 1, (a, b) -> a + b);
System.out.println(authorBooks.get("AuthorB"));// 1
authorBooks.merge("AuthorB", 1, (a, b) -> a + b);
System.out.println(authorBooks.get("AuthorB"));//2
```

11)Method : V putIfAbsent(K key, V value)

class/interface: Map

Example

if the key is not present or if the key is mapped to null, then the key-value pair is added to the map and the result returns null. If the key is already mapped to a value, then that value is returned

```
System.out.println(authorBooks.putIfAbsent("AuthorC", 2));//null
System.out.println(authorBooks.putIfAbsent("AuthorC", 2));//2
```

12)Method : boolean remove(Object key, Object value)

class/interface: Map

Example

removes the key only if its associated with the given value

13)Method : V replace(K key, V newValue)

class/interface: Map

Example

If the key is present then the value is replaced by newValue. If the key is not present, does nothing.

14)Method : boolean replace(K key, V oldValue, V newValue)

class/interface: Map

Example

If the key is present and is mapped to the oldValue, then it is remapped to the newValue.

15)Method : void replaceAll(BiFunction<? super K, ? super V,? extends V> function)

class/interface: Map

Example

replaces all values by the values computed from this function.

```
authorBooks.replaceAll((a,b)->a.length()+b);
```

replaces the count of books by the letters in authors words + original count

How linked list is replaced with binary tree?

In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

This JDK 8 change applies only to **HashMap, LinkedHashMap** and **ConcurrentHashMap**.

Based on a simple experiment of creating HashMaps of different sizes and performing put and get operations by key, the following results have been recorded.

1. HashMap.get() operation with proper hashCode() logic

Number Of Records	Java 5	Java 6	Java 7	Java 8
10,000	4 ms	3 ms	4 ms	2 ms
100,000	7 ms	6 ms	8 ms	4 ms
1,000,000	99 ms	15 ms	14 ms	13 ms

- **Concurrency API improvement**

New classes and interfaces in java.util.concurrent
The java.util.concurrent package contains two new interfaces and four new classes:

- Interface CompletableFuture.AsynchronousCompletionTask: A marker interface identifying asynchronous tasks produced by async methods.
- Interface CompletionStage<T>: A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes.
- Class CompletableFuture<T>: A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.
- Class ConcurrentHashMap.KeySetView<K,V>: A view of a ConcurrentHashMap as a Set of keys, in which additions may optionally be enabled by mapping to a common value.
- Class CountedCompleter<T>: A ForkJoinTask with a completion action performed when triggered and there are no remaining pending actions.
- Class CompletionException: Exception thrown when an error or other exception is encountered in the course of completing a result or task.

New methods in java.util.concurrent.ConcurrentHashMap
The Collections Framework has undergone a major revision in Java 8 to add aggregate operations based on the newly added streams facility and lambda expressions. As a result, the ConcurrentHashMap class introduces over 30 new methods in this release. These include various forEach methods (forEach, forEachKey, forEachValue, and forEachEntry), search methods

(search, searchKeys, searchValues, and searchEntries) and a large number of reduction methods
(reduce, reduceToDouble, reduceToLong etc.)

Other miscellaneous methods (mappingCount and newKeySet) have been added as well. As a result of the JDK 8
changes, ConcurrentHashMaps (and classes built from them) are now more useful as caches. These changes include methods to compute values for keys when they are not present, plus improved support for scanning (and possibly evicting) entries, as well as better support for maps with large numbers of elements.

New classes in java.util.concurrent.atomic
Maintaining a single count, sum, etc. that is updated by possibly many threads is a common scalability problem. This release introduces scalable updatable variable support through a small set of new classes (DoubleAccumulator,DoubleAdder, LongAccumulator, LongAdder), which internally employ contention-reduction techniques that provide huge throughput improvements as compared to Atomic variables. This is made possible by relaxing atomicity guarantees in a way that is acceptable in most applications.

- DoubleAccumulator: One or more variables that together maintain a running double value updated using a supplied function.
- DoubleAdder: One or more variables that together maintain an initially zero double sum.
- LongAccumulator: One or more variables that together maintain a running long value updated using a supplied function.
- LongAdder: One or more variables that together maintain an initially zero long sum.

New methods in java.util.concurrent.ForkJoinPool
A static commonPool() method is now available and appropriate for most applications. The common pool is used by any ForkJoinTask that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use, and reinstated upon subsequent use). Two new methods (getCommonPoolParallelism() and commonPool()) have been added, which return the targeted parallelism level of the common pool, or the common pool instance, respectively.

New class java.util.concurrent.locks.StampedLock
A new StampedLock class adds a capability-based lock with three modes for controlling read/write access (writing, reading, and optimistic reading). This class also supports methods that conditionally provide conversions across the three modes.

CompletableFuture

- CompletableFuture is used for asynchronous programming in Java.
- Asynchronous programming is a means of writing non-blocking code by running a task on a separate thread than the main application thread and notifying the main thread about its progress, completion or failure.
- CompletableFuture implements Future and CompletionStage interfaces and provides a huge set of convenience methods for creating, chaining and combining multiple Futures. It also has a very comprehensive exception handling support.

Future vs CompletableFuture

- Future vs CompletableFuture
- CompletableFuture is an extension to Java’s Future API which was introduced in Java 5.
- A Future is used as a reference to the result of an asynchronous computation. It provides an isDone() method to check whether the computation is done or not, and a get() method to retrieve the result of the computation when it is done.
- Future API was a good step towards asynchronous programming in Java but it lacked some important and useful features -

Limitations of Future

- It cannot be manually completed :
    - Let's say that you've written a function to fetch the latest price of an e-commerce product from a remote API. Since this API call is time-consuming, we're running it in a separate thread and returning a Future from your function.
      Now, let's say that If the remote API service is down, then we want to complete the Future manually by the last cached price of the product.
      Can you do this with Future? No!
- You cannot perform further action on a Future's result without blocking:
    - Future does not notify you of its completion. It provides a get() method which blocks until the result is available.
    - We don't have the ability to attach a callback function to the Future and have it get called automatically when the Future's result is available.
- Multiple Futures cannot be chained together :
    - Sometimes we need to execute a long-running computation and when the computation is done, you need to send its result to another long-running computation, and so on.
    - We can not create such asynchronous workflow with Futures.
- You can not combine multiple Futures together :
    - Let's say that we have 10 different Futures that you want to run in parallel and then run some function after all of them completes. You can't do this as well with Future.

- No Exception Handling :
    - Future API does not have any exception handling construct.

```
import java.util.concurrent.Callable;

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

import java.util.concurrent.TimeUnit;


public class CompletableFutureExample {

	public static void main(String[] args) throws InterruptedException,
ExecutionException {

		ExecutorService executor =
Executors.newFixedThreadPool(1);

		CompletableFuture cf = CompletableFuture.supplyAsync( ( )
-> {


			try {

			for(int i=0;i<1000;i++)
```

```
            {
                System.out.println(i);
            }
          TimeUnit.SECONDS.sleep(1);
          return 123;
        }
        catch (InterruptedException e) {
          throw new IllegalStateException("task interrupted", e);
        }
      },executor);

      Thread.sleep(20);
      cf.complete(" explicitly Done!");
      Object result = cf.get();

      System.out.println("result: " + result);

  }

}
```
____

- **Java IO improvements**
  - Files.list(Path dir) that returns a lazily populated Stream,the elements of which are the entries in the directory
  - Files.lines(Path path) that reads all lines from a file as a Stream
  - Files.find() that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
  - BufferedReader.lines() that return a Stream,the elements of which are lines read from this BufferedReader

- **Nashorn JavaScript Engine**

With Java 8, Nashorn, a much improved javascript engine is introduced to replace the existing Rhino java script engine. Nashorn provides 2 to 10 times better performance as it directly compiles the code in memory and passes the bytecode to JVM.Nashorn uses invoke dynamics feature, introduced in java 7 to improve performance.

- jjs

  For Nashorn engine, JAVA 8 introduces a new command line tool, jjs to execute java script code at console.

- Interpretting js file.

  Create and save sample.js in **c: > JAVA** folder.

sample.js

```
print('Hello World!');
```

Open console and use the following command.

```
C:\JAVA>jjs sample.js
```

See the result.

```
Hello World!
```

- jjs in interactive mode

  Open console and use the following command.

```
C:\JAVA>jjs
jjs> print("Hello, World!")
Hello, World!
jjs> quit()
```

```
>>
```

- pass arguments

Open console and use the following command.

```
C:\JAVA> jjs -- a b c
jjs> print('letters: ' +arguments.join(", "))
letters: a, b, c
jjs>
```

- Invoking Javascript Functions from Java

Nashorn supports the invocation of javascript functions defined in your script files directly from java code. You can pass java objects as function arguments and return data back from the function to the calling java method.

The following javascript functions will later be called from the java side:

```
var fun1 = function(name) {
   print('Hi there from Javascript, ' + name);
   return "greetings from javascript";
};

var fun2 = function (object) {
   print("JS Class Definition: " +
Object.prototype.toString.call(object));

};
```

In order to call a function you first have to cast the script engine to Invocable. The Invocable interface is implemented by the NashornScriptEngine implementation and defines a methodinvokeFunction to call a javascript function for a given name.

```
ScriptEngine engine = new
ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new FileReader("script.js"));

Invocable invocable = (Invocable) engine;

Object result = invocable.invokeFunction("fun1", "Peter Parker");
System.out.println(result);
System.out.println(result.getClass());

// Hi there from Javascript, Peter Parker
// greetings from javascript

// class java.lang.String
```

Executing the code results in three lines written to the console. Calling the function `print` pipes the result to `System.out`, so we see the javascript message first.

Now let's call the second function by passing arbitrary java objects:

```
invocable.invokeFunction("fun2", new Date());
// [object java.util.Date]

invocable.invokeFunction("fun2", LocalDateTime.now());
// [object java.time.LocalDateTime]

invocable.invokeFunction("fun2", new Person());
// [object com.winterbe.java8.Person]
```

- Invoking Java Methods from Javascript

Invoking java methods from javascript is quite easy. We first define a static java method:

```
static String fun1(String name) {
   System.out.format("Hi there from Java, %s", name);
   return "greetings from java";
}
```

Java classes can be referenced from javascript via the Java.type API extension. It's similar to importing classes in java code. As soon as the java type is defined we naturally call the static method fun1() and print the result to sout. Since the method is static, we don't have to create an instance first.

```
var MyJavaClass = Java.type('my.package.MyJavaClass');

var result = MyJavaClass.fun1('John Doe');
print(result);

// Hi there from Java, John Doe
// greetings from java
```

- **Class dependency analyzer: jdeps**

  jdeps is a really great command line tool. It shows the package-level or class-level dependencies of Java class files. It accepts **.class** file, **a directory**, or **JAR file** as an input. By default, **jdeps** outputs the dependencies to the system output (console).

  As an example, let us take a look on dependencies report for the popular Spring Framework library. To make example short, let us analyze only one JAR file: **org.springframework.core-3.0.5.RELEASE.jar**.

```
> jdeps org.springframework.core-3.0.5.RELEASE.jar
```

This command outputs quite a lot so we are going to look on the part of it. The dependencies are grouped by packages. If dependency is not available on a classpath, it is shown as **not found.**

```
org.springframework.core-3.0.5.RELEASE.jar
-> C:\Program
Files\Java\jdk1.8.0\jre\lib\rt.jar
   org.springframework.core
(org.springframework.core-
3.0.5.RELEASE.jar)
      -> java.io
      -> java.lang
      -> java.lang.annotation
      -> java.lang.ref
      -> java.lang.reflect
      -> java.util
      -> java.util.concurrent
      -> org.apache.commons.logging
not found
      -> org.springframework.asm
not found
      -> org.springframework.asm.commons
not found
   org.springframework.core.annotation
(org.springframework.core-
3.0.5.RELEASE.jar)
      -> java.lang
      -> java.lang.annotation
      -> java.lang.reflect
      -> java.util
```

- **Base64**

Finally, the support of Base64 encoding has made its way into Java standard library with Java 8 release. It is very easy to use as following example shows off

```
import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class Base64Test {

    public static void main(String[] args) {
        final String text = "Base64 finally in Java
8!";
        final String encoded =
Base64.getEncoder().encodeToString( text.getBytes(
StandardCharsets.UTF_8 ) );
        System.out.println( encoded );

         final String decoded = new
String(Base64.getDecoder().decode( encoded
),StandardCharsets.UTF_8 );
            System.out.println( decoded );

    }
}
```

The console output from program run shows both encoded and decoded text:

```
QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==
Base64 finally in Java 8!
```

There are also URL-friendly encoder/decoder and MIME-friendly encoder/decoder provided by the Base64 class (Base64.*getUrlEncoder*() / Base64.*getUrlDecoder*(), Base64.*getMimeEncoder*() /Base64.*getMimeDecoder*()).

- **New Features in Java compiler- Parameter names**

With Java 8 and the compiler flag: javac -parameters method parameter names are available via reflection. For example: the parameter names of the method hello:

```
public class Demo{

   public void hello(String name, int age) {

   }
}
   public static void main(String[] args) {
      Method[] methods = Demo.class.getMethods();
      for (Method method : methods) {
         System.out.print(method.getName() + "(");
         Parameter[] parameters = method.getParameters();
         for (Parameter parameter : parameters) {
            System.out.print(parameter.getType().getName() + " "
+ parameter.getName() + " ");
```

```
            }
            System.out.println(")");
        }
    }
}
```

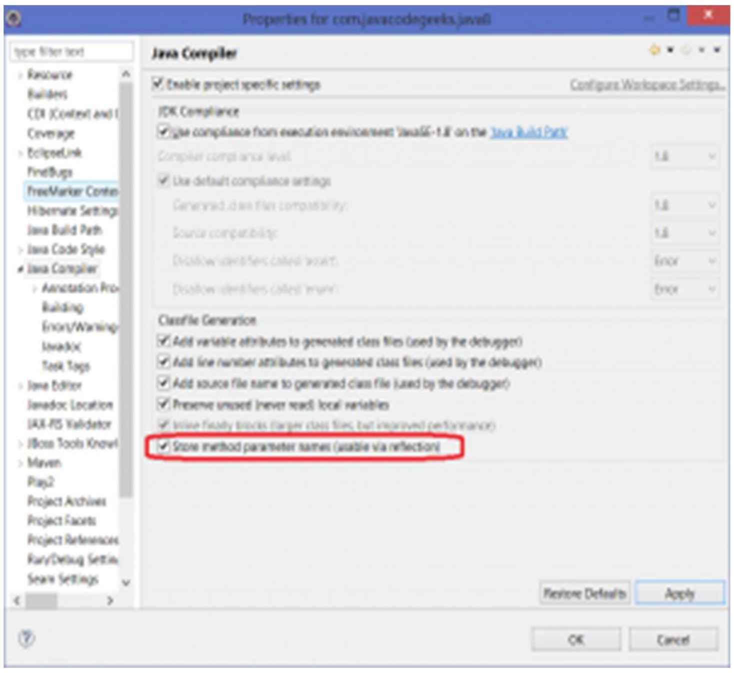
Compilation with javac -parameters produces the following output:

hello(java.lang.String **name** int **age**)

Without the -parameters flag the names are not available:

hello(java.lang.String **arg0** int **arg1**)

Latest Eclipse release with Java 8 support provides useful configuration option to control this compiler setting as the picture below shows.



Properties for com.javacodegeeks.java8

type filter text

Resource
Builders
CDI (Context and I
Coverage
EclipseLink
FindBugs
FreeMarker Conte
Hibernate Settings
Java Build Path
Java Code Style
Java Compiler
Annotation Pro
Building
Errors/Warnings
Javadoc
Task Tags
Java Editor
Javadoc Location
JAX-RS Validator
JBoss Tools Knowl
Maven
Play2
Project Archives
Project Facets
Project References
Run/Debug Settin
Seam Settings

Java Compiler

☑ Enable project specific settings — Configure Workspace Settings...

JDK Compliance

☑ Use compliance from execution environment 'JavaSE-1.8' on the Java Build Path

Compiler compliance level: 1.8

☑ Use default compliance settings

Generated .class files compatibility: 1.8

Source compatibility: 1.8

Disallow identifiers called 'assert': Error

Disallow identifiers called 'enum': Error

Classfile Generation

☑ Add variable attributes to generated class files (used by the debugger)

☑ Add line number attributes to generated class files (used by the debugger)

☑ Add source file name to generated class file (used by the debugger)

☑ Preserve unused (never read) local variables

☑ Inline finally blocks (larger class files, but improved performance)

☑ Store method parameter names (usable via reflection)

Restore Defaults | Apply

OK | Cancel

- **New Features in Java runtime (JVM)**

One important change in Memory Management in Java 8

Oracle's latest edition for Java – Java 8 was released in March 2014. As usual, tons of new features have been added. There is one major change in the Memory management area

So long PermGen, Hello Metaspace !!"

Oracle has completely gotten rid of 'PermGen' and replaced it with Metaspace.

What is PermGen ?
Short form for Permanent Generation, PermGen is the memory area in Heap that is used by the JVM to **store class and method objects**. If your application loads lots of classes, PermGen utilization will be high.
PermGen also **holds 'interned' Strings**
The size of the PermGen space is configured by the Java command line option **-XX:MaxPermSize**
Typically 256 MB should be more than enough of PermGen space for most of the applications

However, It is not unusal to see the error
"**java.lang.OutOfMemoryError: PermGen space"** if you are loading unusual number of classes.
Gone are the days of OutOfMemory Errors due to PermGen space.
With Java 8, there is NO PermGen. That's right. So no more OutOfMemory Errors due to PermGen
The key difference between PermGen and Metaspace is this: while PermGen is part of Java Heap (Maximum size configured by -Xmx option), **Metaspace is NOT part of Heap.** Rather Metaspace is part

of **Native Memory (process memory)** which is only limited by the Host Operating System.

Java Type and Repeating Annotations

Java Type Annotations

Java 8 has included two new features repeating and type annotations in its prior annotations topic. In early Java versions, you can apply annotations only to declarations. After releasing of Java SE 8 , annotations can be applied to any type use. It means that annotations can be used anywhere you use a type. For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this:

1. @NonNull String str;

Following are the examples of type annotations:

1. @NonNull List<String>
1. List<@NonNull String> str
1. Arrays<@NonNegative Integer> sort
1. @Encrypted File file
1. @Open Connection connection
1. void divideInteger(int a, int b) throws @ZeroDivisor ArithmeticException

Note - Java created type annotations to support improved analysis of Java programs. It supports way of ensuring stronger type checking.

Java Repeating Annotations

In Java 8 release, Java allows you to repeating annotations in your source code. **It is helpful when you want to reuse annotation for the same class.** You can repeat an annotation anywhere that you would use a standard annotation.

For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. In order for the compiler to do this, two declarations are required in your code.

1. Declare a repeatable annotation type
2. Declare the containing annotation type

---

1) Declare a repeatable annotation type

Declaring of repeatable annotation type must be marked with the @Repeatable meta-annotation. In the following example, we have defined a custom @Game repeatable annotation type.

```
@Repeatable(Games.class)
@interface Game{
    String name();
    String day();
}
```

The value of the @Repeatable meta-annotation, in parentheses, is the type of the container annotation that the Java compiler generates to store repeating annotations. In the following example, the containing annotation type is Games. So, repeating @Game annotations is stored in an @Games annotation.

---

2) Declare the containing annotation type

Containing annotation type must have a value element with an array type. The component type of the array type must be the repeatable annotation type. In the following example, we are declaring Games containing annotation type:

```
@interface Games{
    Game[] value();
}
```

Note - Compiler will throw a compile-time error, if you apply the same annotation to a declaration without first declaring it as repeatable.

Java Repeating Annotations Example

```
// Importing required packages for repeating annotation

import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

// Declaring repeatable annotation type
@Repeatable(Games.class)
@interface Game{
   String name();
        String day();
     }

     // Declaring container for repeatable annotation type

     @Retention(RetentionPolicy.RUNTIME)
     @interface Games{
        Game[] value();
     }
```

```
// Repeating annotation
@Game(name = "Cricket",  day = "Sunday")
@Game(name = "Hockey",   day = "Friday")
@Game(name = "Football", day = "Saturday")
public class RepeatingAnnotationsExample {
    public static void main(String[] args) {
        // Getting annotation by type into an array
        Game[] game = RepeatingAnnotationsExample.class.getAnnotationsByType(Game.class);
        for (Gamegame2 : game) {    // Iterating values
            System.out.println(game2.name()+" on "+game2.day());
        }
    }
}
```

OUTPUT:

```
Cricket on Sunday
Hockey on Friday
Football on Saturday
```