Java8 features

Date-07/10/24

I) default and static methods in an interface

- Before Java8, an interface can contain only public static final variables and public abstract methods.
- Suppose, an interface has multiple implementation classes in a project.
- Now, due to a change in the requirements, new abstract methods are added to the interface.
- Because of this new abstract methods, all the implementation classes will get an error.
- Before Java8, the solution is, reach out each and every implementation class and override the new abstract methods also in that class.
- It will take more time, and hence the productivity is reduced.
- From Java8, we can write default methods in an interface.
- So, now instead of adding new abstact methods, we can define default methods in the interface.
- default method is created with default keyword and it can have the body in the interface.
- A class can use the default implementation or it can override with its own implementation.

How to create a default method?

```
interface MyInter {

    void m1();

    default void m2() {

        //default implementaion

    }

}
```

```
/*
 * This class is only overriding
 * abstract method m1()
*/
class MyClass1 implements MyInter {
    @Override
    public void m1() {
        //logic
    }
 }

 /*
 *This class is overriding the
 *abstract method and also the
 *default method
*/
 class MyClass2 implements MyInter {
    @Override
    public void m1() {
         //logic
    }
    @Override
    public void m2() {
         //logic
     }
```

```
}
```

- Like default methods, suppose you want to share some common functionality with the implementation classes and if you don't want to allow the classes to override/redefine the functionality, then you have to create static methods in interface.

ex:

```
public interface MyInter {
      void m1();  //abstract method
      default void m2() {
            //default implementation
      }
      static void m3() {
            //common implementation
      }
}
```

II) Functional interface and lambda expression

- A functional interface is an interface with a single abstract method.
- A functional interface may contain any number of variables(public static final), default methods and static methods. But it must contain only one abstract method.

ex1:

```
interface MyInter {
      void m1();
      default void m2() {
          //default logic
      }
```

```
    }
```

- The above interface is a functional interface.

ex2:

```
   interface MyInter {
      void m1();
      default void m2() {
           //default logic
       }
      default void m3() {
           //default logic
      }
    }
```

- The above interface is a functional interface.

ex3:

```
  interface MyInter {
       void m1();
       void m2();
       default void m3() {
            //default logic
       }
       static void m4() {
            //common logic
       }
```

```
}
```

- The above interface is not a functional interface. Just it is a normal interface.

ex4:

```
@FunctionalInterface
interface MyInter {
    void m1();
    void m2();
    default void m3() {
        //default logic
    }
    static void m4() {
        //common logic
    }
}
```

- The above example is a compile time error. Because of @FunctionalInterface.

Ex5:

```
@FunctionalInterface
pubic interface MyInter {
    void m1();
    String toString();
    default void m2() {
```

```
            //default logic
        }
    }
```

- The above interface is a functional interface.
- If a method of Object class is declared as an abstract method in an interface which has a single abstract method, then it is a functional interface only.

Ex6:

```
@FunctionalInterface
public interface MyInterface {
    void m1();
    boolean equals(Object o);
    int hashCode();
}
```

- The above interface is a functional interface.
- Actually it has 3 abstract methods, but 2 of them are matching with methods of Object class. So, this interface has only one abstract method.

Some pre-defined functional interfaces are,

Runnable	void run()
Comparable<T>	int compareTo(T t)
Comparator<T>	int compare(T o1, T o2)
Predicate<T>	boolean test(T t)
Consumer<T>	void accept(T t)
Supplier<T>	T get()
Function<T>	R apply(T t)

---

lambda expressions:

- suppose, we have a functional interface, lets say Comparator.
- I have a requirement to sort the list of employees in ascending

order of emp numbers.
what to do?
. first, create a class for implementing Comparator interface to sort the employees in ascending order of emp numbers.
. next, sort the employees list using the Comparator object.

- I have a requirement to sort the list of employees in ascending order of employee salaries.
  what to do?
  . first, create a class for implementing Comparator interface to sort the employees in ascending order of emp salaries.
  . next, sort the employees list using the Comparator object.
- Like this, if I have multiple requirements, then I have to create multiple implementation classes for Comparator interface.
- It will increase number of classes in the project.
- So, to reduce the number of classes, Java8 has provided a solution called lambda expressions, to provide the implementation for functional interfaces, by without creating a class.

For example:

with class:

```
class EmpnoComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
}
```

with lambda expression:

```
(e1, e2) -> e1.getEmpno() - e2.getEmpno();
```

- I want to sort the list of employees in ascending order of emp numbers.

```
Collections.sort(lstEmp, new EmpnoComparator());
```

with lambda,

```
Collections.sort(lstEmp, (e1,e2) -> e1.getEmpno() - e2.getEmpno());
```

- I want to sort the list of employees in ascending order of emp salaries.

```
Collections.sort(lstEmpo, new SalComparator());
```

with lambda,

```
Collections.sort(lstEmp, (e1,e2)-> e1.getSal() - e2.getSal());
```

- we can use lambda expressions, only for providing implementation for functional interfaces. But not for normal interfaces.
- syntax:
  (arguments) -> body
  The arguments should match with the parameters of the abstract method of the functional interface.
- The body of the lambda expression can have a single or multiple statements. For a single statement, curly braces is optional.

example1:

```
@FunctionalInterface
public interface MyFunctional {
     int add(int a, int b);
}
```

The lambda expression to implement the above interface is,

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

```
        }
```

- arguments in lambda expression must be same as the arguments of the abstract method.
- arguments names can be different, but the number of arguments, type of arguments and the order of the arguments must be same.
- if the arguments are zero or more than one then paranthesis is mandatory. For one argument paranthesis is optional.
- Type of the arguments is optional. But don't specify the type for one argument and not for the other arguments.

examples:

```
1. (int a, int b)  -> System.out.println(a+b); //correct
2. (a,b) -> System.out.println(a+b); //correct
3. (int a, b) -> System.out.println(a+b); //error
4. a, b -> System.out.println(a+b); // error
5.  a -> System.out.println(a * a); //correct
6. () -> System.out.println( "hello" ); //correct
```

sort(Comparator<T> c) & forEach(Consumer<T> c) :

- From Java8, sort() method is included as a default method in List interface. So, we can sort the elements of a List object, by calling this sort() method. There is no need to use Collections.sort() statement from Java8.
- The sort() method of the List interface accepts Comparator object as an argument.
- Comparator is a functional interace. So, if any method accepts an object of type functional interface as a parameter, then we can pass lambda expression as an argument.
- From Java8, forEach() method is included as a default method in the Iterable interface. So, we can iterate the collection with forEach method also, instead of using Iterator or for each loop.
- The forEach() method accepts Consumer object as an argument.
- Consumer is a functional interface. So, we can pass lambda expression as an argument.

```
package pack1;

import java.util.ArrayList;
import java.util.List;

class Employee  {
    private int empno;
    private String ename;
    private double sal;
    private String gender;
    private double experience;

    public Employee(int empno, String ename, double sal, String gender, double experience)
{
            super();
```

```
        this.empno = empno;
        this.ename = ename;
        this.sal = sal;
        this.gender = gender;
        this.experience = experience;
}

public int getEmpno() {
        return empno;
}

public void setEmpno(int empno) {
        this.empno = empno;
}

public String getEname() {
        return ename;
}

public void setEname(String ename) {
        this.ename = ename;
}

public double getSal() {
        return sal;
}

public void setSal(double sal) {
        this.sal = sal;
}

public String getGender() {
        return gender;
}

public void setGender(String gender) {
        this.gender = gender;
}

public double getExperience() {
        return experience;
}

public void setExperience(double experience) {
        this.experience = experience;
}

@Override
public String toString() {
```

```
            return "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ",
gender=" + gender + ", experience="
                            + experience + "]";
    }
}


public class Solution {

    public static void main(String[] args) {
            List<Employee> empList = new ArrayList<>();
            empList.add(new Employee(7298, "Scott", 5000.0, "Male", 4.5));
            empList.add(new Employee(7178, "Allen", 7000.0, "Male", 5.5));
            empList.add(new Employee(7154, "Kathey", 6000.0, "Female", 4.5));
            empList.add(new Employee(7233, "Clark", 5000.0, "Male", 4.5));
            empList.add(new Employee(7741, "Mary", 4000.0, "Female", 3.1));

            //sorting the list in empno ascending order
            empList.sort((e1, e2) -> e1.getEmpno() - e2.getEmpno());

            System.out.println("Displaying the employee in empno ascending order");
            empList.forEach( e -> System.out.println(e));
          // Consumer<Employee> cons = e -> System.out.println(e);
          // empList.forEach(cons);

    }

}
```

III) Optional<T> class

The most commonly raised exception in java application is NullPointerException.

The reason, we call a method on an object, and if that object has null value, then NullPointerException will be raised at runtime.

To avoid NullPointerException, we have to write the code with null checks.

For example:

```
if(student!=null) {

   Address addr = student.getAddress();

   if(addr != null) {

     Country ct = addr.getCountry();

     if(ct != null) {

        State st = ct.getState();

        if( st !=null) {
```

```
            City c = st.getCity();
            if( c != null) {
               String name = c.getName();
            }
         }
      }
   }
```

- If we add more null checks in a code, it increases code complexity and reduces code readability.
- So, to minimize the null checks in application and also to minimize NullPointerExceptions, Java8 has provided Optional class.
- Optional class object is container object, which may or may not contain non-null value.
- Optional class object can be created in 3 ways.
  1. Optional<T> opt = Optional.empty();
  2. Optional<T> opt = Optional.ofNullable(T t);
  3. Optional<T> opt = Optional.of(T t);

For example:

```
Optional<Student> opt = Optional.ofNullable(stu);
```

if stu is null then opt is empty. if stu is not null then opt stores stu object.

```
Optional<Student> opt = Optional.of(stu);
```

if stu is null throws NullPointerException. if stu is not null then opt stores stu object.

- Optional class has methods like isPresent(), get() and ifPresent().
- isPresent() checks for a value in the Optional object. if value exists then returns true, otherwise returns false.
- if value exisits then call get() method to fetch the value from the Optional object.

for example:

```
Optional<Employee> opt = repository.findById(7788);
if( opt.isPresent() ) {
   Employee e = opt.get();
}
```

- ifPresent() method performs the given action, if value is present. Otherwise, do nothing.

for example:

```
Optional<Employee> opt = repository.findById(7788);

opt.ifPresent( e -> System.out.println(e) );

opt.ifPresent( e -> System.out.println(e) ).orElse(S.o.p( "employee not found" ));
```

IV) stream api

- A collection object like a list or a set or map object is used to store the elements.
- if you want to process the elements then you have to use if conditions, loops, iterators,etc.
- If a collection object has more elements then the processing the collection will take more time.
- So, to reduce the time to process the elements of a collection object, Java8 has introducted stream api.
- The purpose of a collection object and a stream object is different. A collection object is to store the elements and a stream object is to process the elements.
- A stream object can be created from different sources.
  1. we can create a stream object from a collection object.
     `Stream<Employee> stream = empList.stream();`
  2. we can create a stream object from an array.
     `String[] names = { "John" ," Jill" ," Jack" ," Jenny" };`
     `Stream<String> stream = Arrays.stream(names);`
  3. we can create a stream from raw values.
     `Stream<Integer> stream = Stream.of(10, 20, 13, 25, 15 );`
  4. we can create an empty stream.
     `Stream<Void> stream =Stream.empty();`
- To process the elements of a Stream, we have two types of operations.
  1. intermediate operations
  2. terminal operations.
- intermediate operation transforms one stream to another stream.
- terminal operation produces the result.

For example:

```
Stream<Employee> stream = empList.stream();

Stream<Employee> stream2 = stream.filter( e -> e.getSalary() > 5000);
```

. Here, filter() is an intermediate operation. Because it is transforming one stream to another .

```
long count = stream2.count();
```

. Here, count() is a terminal opertion. Because it is producing a result not another stream.

- The stream operations like filter(), map(), sorted(), iterate(), peek(), skip(), limit(), flatMap() etc.. are intermediate operations. Because they produces/returns another stream.
- The stream operations like count(), collect(), findFirst(), max(), min(), reduce(), forEach(), etc.. are terminal operations.

ex1:

print the employees of a list with salary >= 5000

```
empList.stream().filter( e -> e.getSal() >= 5000 )
              .forEach(e -> System.out.println(e));
```

ex2:

count the employees of a list with salary >= 5000

```
long count = empList.stream()
                    .filter(e -> e.getSal() >= 5000)
                    .count();
```

ex3:

find the employee of a list with highest salary

```
Optional<Employee> opt = empList.stream()
                    .sorted( (e1,e2) -> (int) (e2.getSal() - e1.getSal()))
                    .findFirst();
//opt.ifPresent(e -> System.out.println(e));
if(opt.isPresent()) {
       Employee e = opt.get();
       System.out.println(e);
}
```

ex3:

find the employees of a list with salary > 4000 and sort them in ascending order and then display.

```
empList.stream().filter( e -> e.getSal() > 4000 )
                .sorted( (e1, e2) -> (int)(e1.getSal() - e2.getSal()))
                .forEach( e -> System.out.println(e));
```

ex4:

collect the employees names of a list of employees and store them in another list.

```
              /*
              Stream<Employee>  stream = empList.stream();
              Stream<String> stream2 = stream.map(Employee::getEname);
              List<String> lst = stream2.collect(Collectors.toList());
              lst.forEach(System.out::println);
              */

              List<String> lst = empList.stream()
```

```
                                                .map(Employee::getEname)
                                                .collect(Collectors.toList());
                lst.forEach(System.out::println);
ex5:
 find the second highest paid employee from a list of employees.
                    empList.stream()
                        .sorted( (e1,e2) -> (int) (e2.getSal() - e1.getSal()))
                        .skip(1)
                        .findFirst()
                    .ifPresent(System.out::println);
ex6:
  checking for an employee with sal > 8000 in the list of employees.
 boolean flag = empList.stream()
                        .anyMatch(Employee::getSal > 8000);
ex7:
  checking for all the employees with sal > 3000 in the list of employees.
 boolean flag = empList.stream()
                        .allMatch(Employee::getSal > 3000);
ex8:
 create a new list for a list of employees, after incrementing their salary by 5000
List<Employee> newList = empList.stream()
                        .map(e -> { e.setSal(e.getSal() + 5000); return e; })
                        .collect(Collectors.toList());
newList.forEach(System.out::println);

ex9:
 print only first 5 highest paid employees
empList.stream()
       .sorted( (e1, e2) -> (int) (e2.getSal() - e1.getSal()))
       .limit(5)
       .forEach(System.out::println);
ex9:
// Collect names into a List, whose length > 3

List<String> names = Arrays.asList("John", "Jane", "Tom", "Doe", "Jill", "Jeffry",
"Jackson");

List<String> collectedNames = names.stream()
                                        .filter(name -> name.length() > 3)
                                        .collect(Collectors.toList());
collectedNames.forEach(System.out::println);

ex10:
// Join names with a comma separator
List<String> names = Arrays.asList("John", "Jane", "Tom", "Doe", "Jeffry");

String joinedNames = names.stream().collect(Collectors.joining(", "));

System.out.println(joinedNames);
```

ex11:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Sum all numbers
int sum = numbers.stream()
                 .collect(Collectors.summingInt(Integer::intValue));

System.out.println(sum);
```

ex12:

Group the employees of a list by their gender value.

```
Map<String, List<Employee>>  map = empList.stream()
                          .collect(Collectors.groupingBy(Employee::getGender));
System.out.println(map);
```

---

parallelStream():

. If the source of data to process is huge, like a list of 10 million employees and if we use a stream then it takes more time to process the elements.

. Because, a normal stream can process the elements sequentially.

. To improve the performance, we have to use parallel stream.

. parallel stream divides the source of data into data chunks and executes the stream operations on each chunk on a separate core of the cpu.

. after processing, it will join the results obtained from each core and returns the final result.

. So, parallel stream utilizes the full potential of the cpu, to perform the operations.

For example:

```
List<Employee> lst = empList.parallelStream()
                            .filter(e -> e.getSal() > 5000)
                            .collect(Collectors.toList());
```

(V) CompletableFuture<T> Class:

1. Executor Framework was added in Java5, to simplify the multithreading and to manage and control the thread execution using a high-level API.
2. Before Java5, programmers have to manually create and manage the threads using Thread class and Runnable interface.
3. The problems here threads, developers have to take care of synchronization manually and creating too many threads will consume more system resources.
4. So, Java5 has introduced Executor Framework, where a developer has to define the task and has to submit it to the Executor Framework. The rest of the work like creating the threads, executing the threads, synchronization, etc… will be take care by Executor Framework only.
5. The key components of Executor Framework only.
   1. Executor interface - Basic interface
   2. ExecutorService interface - extends Executor interface
   3. Executor class - utility class, creates ExecutorService instance.

Types of Executors:
---

1. ExecutorService service = Executors.newSingleThreadsExecutor();

   * Creates a single thread, processes the tasks sequentially.

   * Suppose, if 2 tasks are submitted to the ExecutorService, then first one thread is created to execute tasks1, after that thread dies, another thread is created to execute task2.

2. ExecutorService service = Executors.newFixedThreadPool(n);

   * Creates a fixed number of threads.

   * It is good for executing controlled number of parallel tasks.

3. ExecutorService service = Executors.newCachedThreadPool();

   * Creates new threads as needed and also reuses existing threads.

4. ExecutorService service = Executors.newScheduledThreadPool(n);

   * It is used when you want to execute scheduled tasks, repeatedly at fixed intervals.

How to submit my task to the ExecutorService?

1. void execute (Runnable runnable)

2. Future submit (Callable callable)

   Runnable and Callable are the functional interfaces.

   Runnable' s abstract method - void run();

   Callable' s abstract method - V call() throws Exception.

- Runnable doesn' t return the result and doesn' t throw Exception. But Callable returns the computed result, if unable to compute then throws Exception.
- Future<T> is an interface from Java 5+ and it is to hold the result of a task, going to be generated in future.

   Ex1: public class TestClass {

   public static void main(String[] args) throws Exception
{
        ExecutorService executor = Executors.*newFixedThreadPool*(2);

```
        Callable<String> task = () -> {

                Thread.sleep(5000);
                return "hello".toUpperCase();
        };

        Future<String> future = executor.submit(task);

        System.out.println("I am continuing my work .....");
        System.out.println("My work is completed.....");

        System.out.println("The computed result: " + future.get());

        executor.shutdown();
    }
}
```

Ex2 :

```
public class MainClass {


    public static void main(String[] args) throws Exception {

        Supplier<String> supplier = () -> {
                sleep(5000);
                return "Hello World";
        };

        CompletableFuture<String> future =
CompletableFuture.supplyAsync(supplier);

        System.out.println("I am continuing my work ........");

        future.thenAccept(System.out::println);

        System.out.println("Still my work is going on ......");
        System.out.println("I am done with my work");

        Thread.currentThread().join();
    }

    private static void sleep(long ms) {
```

```
            try {
                    Thread.sleep(ms);
            } catch (InterruptedException ex) {
                    System.out.println(ex);
            }
    }

}
```

Example with thenCombibe() -

```
public class DemoClass {

    public static void main(String[] args) {

            CompletableFuture<String> cf1 = CompletableFuture.supplyAsync(()-
>"Dukhishyam");
            CompletableFuture<String> cf2 = CompletableFuture.supplyAsync(()->" Tudu");

            CompletableFuture<String> cf3 = cf1.thenCombine(cf2, (r1,r2) -> r1 + r2);

            cf3.thenAccept(System.out::println);
    }

}
```

(VI) Spliterator interface

1. The Spliterator is a special-purpose iterator designed for parallel processing of elements in streams and collections.
2. It can split a data source (such as a Collection or Stream) into multiple parts for more efficient parallel execution, hence the name **Split-iterator**.
3. This is especially useful when processing large collections using the Stream API.
4. The key methods of Spliterator interface are,
   tryAdvance() and trySplit()
5. tryAdvance() method will perform the given action on the next element of the spliterator, if exists and returns true. If next elements does not exist then returns false.
6. trySplit() method will split the Spliterator object into two parts.

7. When you are processing large collections, to divide the collection into parts and to perform parallel processing, we use trySplit() method.
8. tryAdvance() method is like a combination of hasNext() method and next() method of Iterator.

ex:

```
List<String>  lst = Arrays.asList("John", "Jack", "Tom", "Jeffry", "Jill",
"Miller", "Allen");

Spliterator<String> split1 = lst.spliterator();

while( split1.tryAdvance(System.out::println));
```

ex:

```
List<String>  lst = Arrays.asList("John", "Jack", "Tom", "Jeffry", "Jill",
"Miller", "Allen");

Spliterator<String> split1 = lst.spliterator();

Spliterator<String> split2 = split1.trySplit();
//Here, split2 processes first half of the collection and
// split1 processes second half of the collection.

System.out.println("printing first half");
split2.forEachRemaining(System.out::println);

System.out.println("printing second half");
split1.forEachRemaining(System.out::println);
```

ex:

```
List<String>  lst = Arrays.asList("John", "Jack", "Tom", "Jeffry", "Jill", "Miller",
"Allen");

Spliterator<String> split1 = lst.spliterator();

Spliterator<String> split2 = split1.trySplit();

Spliterator<String> split3 = split2.trySplit();

Spliterator<String> split4 = split3.trySplit();

if(split4 != null)
       split4.forEachRemaining(System.out::println);

System.out.println("printing first half");
split3.forEachRemaining(System.out::println);

System.out.println("printing second half");
split2.forEachRemaining(System.out::println);
```

```
System.out.println("printing third half");
split1.forEachRemaining(System.out::println);
```

(VII) Date/Time API

1. LocalDate class
2. LocalTime class
3. LocalDateTime class
4. ChronoUnit(enum)
5. Period class

LocalDate class and LocalTime class and LocalDateTime class have private constructor in the class. So, we can not create object for these classes with new keyword.

. These classes have static factory methods like now() and of() for constructing the objects.

For example:

LocalDate date1 = LocalDate.now();

. LocalDate object is created with current system date.

LocalDate date2 = LocalDate.of(2024, 11, 30);

. LocalDate object is created with given date

. LocalDate object stores the date in yyyy-MM-dd format.

. The objects of LocalDate/LocalTime/LocalDateTime are immutable objects. It means, if we make any changes the result will stored in a new object.

For example:

```
LocalDate date2 = LocalDate.of(2024, 11, 30);
System.out.println(date2);
LocalDate date3 = date2.plusWeeks(2);
System.out.println(date3);
```

output:

```
2024-11-30
2024-12-14
```

. On a LocalDate object, we can call factory methods like, plusDays(), plusMonths(), plusWeeks(), plusYears(), minusDays(), minusMonths(), minusWeeks() and minusYears().

finding the difference between two dates:

```
LocalDate date1 = LocalDate.of(2023, 10, 19);
LocalDate date2 = LocalDate.now();
System.out.println("Difference in days : " +
ChronoUnit.DAYS.between(date1, date2)); //366
```

```
            System.out.println("Difference in months : " +
ChronoUnit.MONTHS.between(date1, date2)); //12
            System.out.println("Difference in years : " +
ChronoUnit.YEARS.between(date1, date2)); // 1
```

finding the difference between two times:

```
            LocalTime time1 = LocalTime.of(9, 35, 55);
            LocalTime time2 = LocalTime.now();
            System.out.println("Difference in hours : "+
ChronoUnit.HOURS.between(time1, time2));
            System.out.println("Difference in minutes : "+
ChronoUnit.MINUTES.between(time1, time2));
            System.out.println("Difference in seconds : "+
ChronoUnit.SECONDS.between(time1, time2));
```

. Period class compares the two dates on days, months and years wise. It means, it will not return the differnce total days, or total months.

ex:

```
  LocalDate date1 = LocalDate.of(2023, 10, 19);
  LocalDate date2 = LocalDate.now();
  Period p = Period.between(date1, date2);
  System.out.println(p.getDays()); // 0
  System.out.println(p.getMonths()); // 0
  System.out.println(p.getYears()); // 1
```

converting a string to LocalDate object:

- if a string has value in yyyy-MM-dd format then it can be converted directly to a LocalDate object by calling parse() method.

ex:

```
            String str = "2024-10-12";
            LocalDate date = LocalDate.parse(str);
            System.out.println(date);
```

ex2:

```
String str = "12, December 2024";
        DateTimeFormatter formatter =
                  DateTimeFormatter.ofPattern("dd, MMMM yyyy");
        LocalDate date = LocalDate.parse(str, formatter);
        System.out.println(date);
```