Fw: Reg: Day 5 Java Features Notes and Handson

Oviya P <oviya.p@hcltech.com>
Tue 7/23/2024 10:26 AM
To:oviyazaya.2000@gmail.com <oviyazaya.2000@gmail.com>

Classification: **Personal Use**

From: Oviya P
Sent: Tuesday, January 30, 2024 6:06 PM
To: Oviya P <oviya.p@hcl.com>
Subject: FW: Reg: Day 5 Java Features Notes and Handson

Classification: **Internal**

Java Features Notes

Java 14 - Mar 2020

1. Switch expression

In java 12 - switch with lambda syntax - preview
Java 13 - yield stmt - preview

Java 14 - switch expr is standarised

```
public class Main {
        public static void main(String[] args) {
                String str="Sun";
                String result=switch(str) {
                case "Mon","Wed","Fri" -> "Evendays";
                case "Tues","Thurs","Sat" -> "Odddays";
                default -> {
                        if(str.isEmpty())
                                yield "Please enter correct day";
                        else
                                yield "It is sunday";
                }
                };
                System.out.println(result);
        }
}
```

2. Enhancement in NullPointerException

```
    String s=null;
    s.toLowercase();
```

-XX:+ShowCodeDetailsInExceptionMessages in VM options

```
public class Main {
        public static void main(String[] args) {
```

```
                    Employee e=null;
                    System.out.println(e.getName());
          }
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "com.pack.Employee.getName()"
because "e" is null
          at com.pack.Main.main(Main.java:7)
```

```
class Box{
}
Box b=new Box();
if(b instanceof Box){
}
```

3. Pattern matching for instanceof operator as in preview feature
 - We dont need to cast the object explicitly, we have to just provide corresponding variable along with type of instance object

```
class Person {

}
class Employee1 extends Person {
          int id;
          double salary;
}
public class Main {
          public static void main(String[] args) {
                    Object obj="hello";
                    //lower version
                    if(obj instanceof String) {
                              String s=(String) obj;
                              System.out.println(s.toUpperCase());
                    }

                    //latest version
                    if(obj instanceof String str) {
                              System.out.println(str.toUpperCase());
                    }

                    Person p=new Employee1();  //DMD
                    if(p instanceof Employee1) {
                              Employee1 e1=(Employee1)p;
                    }
                    if(p instanceof Employee1 e2) {

                    }
     if(p instanceof Employee1 e3 && e3.salary > 10000) {

                    }
          }
}
```

4. Records - preview feature

Java 15 - on Sep 2020

1. Hidden classes
   - cannot be used directly by the bytecode of other classes. It is used by frameworks that generate classes at runtime and use them indirectly using Reflection API
   - It is dynamically created at runtime using Lookup API

```
public class Main {
        public static void main(String[] args) throws Exception {
          MethodHandles.Lookup lookup=MethodHandles.lookup();
          Class<?> hiddenClass=lookup.defineHiddenClass(new byte[0], true).lookupClass();
        }
}
```

2. Records
   - To store couple of values in Java class, we have to unnecessarly create constructors, getter and setter methods, toString(), equals(), hashCode()
   - To remove all these boilerplate code we have to define records and use that to store the data
   - Record is a final class and all variables inside records are final, so we can create immutable class

1. Create Student class

```
public class Student {
  String name;
  String email;
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public String getEmail() {
        return email;
}
public void setEmail(String email) {
        this.email = email;
}
public Student(String name, String email) {
        super();
        this.name = name;
        this.email = email;
}
public Student() {
        super();
        // TODO Auto-generated constructor stub
}
@Override
public String toString() {
        return "Student [name=" + name + ", email=" + email + "]";
}

}
```

2. Create a record

```
public record StudentRecord(String name, String email) {

}
```

This will used to store all value and it will have constructor, toString(), getter(), equals(), hashcode

3. Create main class

```
public class Main {
        public static void main(String[] args) throws Exception {
          Student st1=new Student("Ram",ram@gmail.com);
          StudentRecord sr=new StudentRecord("Sam",sam@gmail.com);
        }
}
```

When we use record, by default all instance variables declared in record as final so it will act as immutable object so we wont be able to create setter method

4. Whenever we create a class, by default the default constructor is created. But in record it will create constructor with parameters and that is called as a canonical constructor
   If we create default constructor it will be error

```
public record StudentRecord(String name, String email) {

        public StudentRecord(String name, String email) { //Canonical constrctor
                this.name=name;
                this.email=email;
        }
        /*public StudentRecord() { //error

        }*/
}
```

5. Like class we can create instance method inside record

```
public record StudentRecord(String name, String email) {

        public void print() {
                System.out.println("Records");
        }
}

public class Main {
        public static void main(String[] args) throws Exception {
          Student st1=new Student("Ram",ram@gmail.com);
          StudentRecord sr=new StudentRecord("Sam",sam@gmail.com);
          sr.print();
        }
}
```

6. We can also create static methods inside records

```
public record StudentRecord(String name, String email) {

        public void print() {
                System.out.println("Records");
        }

        public static void print1() {
                System.out.println("static method");
        }
}

public class Main {
        public static void main(String[] args) throws Exception {
         Student st1=new Student("Ram",ram@gmail.com);
         StudentRecord sr=new StudentRecord("Sam",sam@gmail.com);
         sr.print();
         StudentRecord.print1();
        }
}
```

7. We can create static variables but we cant create instance variables inside records

```
public record StudentRecord(String name, String email) {

        public static String s1="hello";
        //private String s2; //error
        public void print() {
                System.out.println("Records");
        }

        public static void print1() {
                System.out.println("static method");
        }
}
```

8. In the class, normal getter() used to retrieve the data but in records it is directly variable name

```
public class Main {
        public static void main(String[] args) throws Exception {
         Student st1=new Student("Ram",ram@gmail.com);
         StudentRecord sr=new StudentRecord("Sam",sam@gmail.com);
         sr.print();
         StudentRecord.print1();
         System.out.println(st1.getName()+" "+st1.getEmail());
         System.out.println(sr.name()+" "+sr.email());
        }
}
```

9. By default record will have toString()

```
public class Main {
        public static void main(String[] args) throws Exception {
         Student st1=new Student("Ram",ram@gmail.com);
         System.out.println(st1);
```

```
                 StudentRecord sr=new StudentRecord("Sam",sam@gmail.com);
                 System.out.println(sr);
                 sr.print();
                 StudentRecord.print1();
                 System.out.println(st1.getName()+" "+st1.getEmail());
                 System.out.println(sr.name()+" "+sr.email());
                }
}
```

10. Record will not be inherited because by default it is final, but it can be implemented by an interface

```
interface A {
        void show();
}
public record StudentRecord(String name, String email) implements A{

        public static String s1="hello";
        //private String s2; //error
        public void print() {
                System.out.println("Records");
        }

        public static void print1() {
                System.out.println("static method");
        }

        @Override
        public void show() {
                System.out.println("Overridden method");
        }

}
```

11. we can create different type of constructors

12. We also have compact constructor which is only available for records, where we define the constructor and what is the validation that we want to do and after this it will by default set the values

```
public record StudentRecord(String name, String email){
  public StudentRecord(String name, String email) {  //Compact constructor
         if(name.isBlank()) {
                 throw new IllegalArgumentException("Name should not be blank");
         }
         this.name=name;
         this.email=email;
  }

}


public class Main {
        public static void main(String[] args) throws Exception {
         Student st1=new Student("Ram",ram@gmail.com);
         System.out.println(st1);
         StudentRecord sr=new StudentRecord("Sam",sam@gmail.com);
         System.out.println(sr);
         StudentRecord sr1=new StudentRecord(" ",abc@gmail.com);
         System.out.println(sr1);
        }
}
```

```
abstract class A{
  void show();
  void show1() {
  }
}
```

3. Sealed classes

- When it comes to abstract class, we can access the methods by inheriting. When it comes to final class, we cant inherit the class
- Now we want to have inheritance but not every class to do it, there should be limited class which we want to be inherited. So Java introduce new feature called Sealed classes
- Sealed classes can mention which are the subclasses or subinterfaces which can be inherit from the particular class and interface by using "sealed" keyword

Consider we have classes A,B,C,D, now we want to make only B,C class to inherit A class and not other classes

1. We can make class A to be final, then none of the classes can inherit A class

```
final class A {
}
class B extends A{   //error
}
class C extends A{   //error
}
class D{
}
public class Main {
       public static void main(String[] args) throws Exception {
       }
}
```

2. We make class A to be sealed so that it allows only particular class to be inherited using "permits" keyword

```
sealed class A permits B,C {
}
class B extends A{
}
class C extends A{
}
class D{
}
public class Main {
       public static void main(String[] args) throws Exception {
       }
}
```

- If we are permitting B and C class then the sealed subclasses B and C can be defined as sealed or non-sealed or final

```
sealed class A permits B,C {
}
sealed class B extends A{
}
sealed class C extends A{
}

sealed class A permits B,C {
}
non-sealed class B extends A{
}
non-sealed class C extends A{
}

sealed class A permits B,C {
}
final sealed class B extends A{
}
final sealed class C extends A{
}


sealed class A permits B,C {
}
final class B extends A{
}
final class C extends A{
}
class D extends A { //error
}
public class Main {
        public static void main(String[] args) throws Exception {
        }
}

sealed class A permits B,C {
}
sealed class B extends A permits D{
}
final class C extends A{
}
final class D extends B {
}
public class Main {
        public static void main(String[] args) throws Exception {
        }
}
```

- Basically subclasses of sealed class (ie)B,C it should be either final or sealed or non-sealed, so if we make B class to be sealed then it will ask for subclasses, if we make B class to non-sealed anyone can extend this class, final no one can extends that class

```
sealed class A permits B,C {
}
non-sealed class B extends A {
}
final class C extends A{
}
final class D extends B {
}
public class Main {
        public static void main(String[] args) throws Exception {
        }
}
```

- Before permits also, A class can extends or implements anything, always permits should be last thing after extends or implements

```
sealed class A extends Thread implements Cloneable permits B,C {
}
non-sealed class B extends A {
}
final class C extends A{
}
final class D extends B {
}
public class Main {
        public static void main(String[] args) throws Exception {
        }
}
```

- We can also permits the interface and the permitted interface can be either sealed or non-sealed

```
sealed interface X permits Y{}
//sealed interface Y extends Z{}
non-sealed interface Y {}
interface Z{}
sealed class A extends Thread implements Cloneable permits B,C {
}
non-sealed class B extends A {
}
final class C extends A{
}
final class D extends B {
}
public class Main {
        public static void main(String[] args) throws Exception {
        }
}
```

Java 17 - Sep 2021 - LTS

1. Switch expression provided with pattern matching, guarded pattern, null cases - preview feature

a. Pattern matching - used to match pattern in case label (or) we can pass object in switch condition and can be checked for different types

```
public class Main {
    static float getType(Object o){
       return switch(o){
          case Integer i -> i.floatValue();
          case Double d -> d.floatValue();
          case String s -> Float.parseFloat(s);
          default ->  0f;
       };
    }
        public static void main(String[] args) {
       System.out.println("Integer: "+getType(12));
       System.out.println("Double: "+getType(12.23));
       System.out.println("String: "+getType("12.23"));
        }
}
```

```
C:\Training>javac Main.java --release 17 --enable-preview
Note: Main.java uses preview features of Java SE 17.
Note: Recompile with -Xlint:preview for details.

C:\Training>java --enable-preview Main
Integer: 12.0
Double: 12.23
String: 12.23
```

b. Guarded pattern - use to check additional checks

```
public class Main {
    static float getType(Object o){
       return switch(o){
          case String s && s.length() > 2 -> Float.parseFloat(s);
          default ->  0f;
       };
    }
        public static void main(String[] args) {
       System.out.println("String: "+getType("1"));
        }
}
```

3. Null cases - if we want check null in switch smt

```
public class Main {
    static String getType(Object o){
       return switch(o){
          case null -> "Null value";
          case String s  -> "It is a string";
          default ->  "Unknown";
       };
    }
        public static void main(String[] args) {
       System.out.println("String: "+getType("1"));
```

```
            System.out.println("String: "+getType(null));
             }
}
```

Java 19
1. Record pattern with instanceof

```
public record Student(int x,int y){
}

private void print(Object o) {
     if(o instanceof Student s) {
            SOP("x="+s.x()+" y= "+s.y());
      }
}
```

Subject: RE: Reg: Day 4 Java Features Notes and Handson

Classification: **Internal**

Given:

```
public class Employee {

   private final String firstName;
   private final String surname;
   private final LocalDate dateOfBirth;
   private final Company company;
   private final BigDecimal salary;
   private final Address homeAddress;
}
```

1. Find whether there are two employees with the same first name and surname and return the name
2. Find the total number of groups of at least 5 employees living close to each other
  consider all employees with the same 2 first characters of the home address post code a single group. you can collect to map and then stream over it, however the solution has to be a single statement

Java 11 (LTS - Long term support) - Sep 2018

1. JRE updates
     - From Java11 onwards there is no JRE install in user machine, JDK itself contains Java development software + JRE

- From Java11 onwards we can execute java file directly without compiling

```
>java A.java
```

- In single java program, we can provide more than one class with main(), then JVM will search for main() in the order we have provided (ie) top to bottom

```
class C {
   public static void main(String[] args) {
      System.out.println("C-main");
   }
}
class A {
   public static void main(String[] args) {
      System.out.println("A-main");
   }
}
class B {
   public static void main(String[] args) {
      System.out.println("B-main");
   }
}
```

- Java 11 supports backward compatability

- We do not have .class file, JVM first loads all classes into JVM and then at last it will load our class
>java -verbose Test.java

- We must place main() in first class, so if first class dosent have main() it leads to compilation error
- No need to have class name and java file name to be same, even if it is public, because java cmd is not verifying the class and java file name is same or not, it just check the file and check first class contains main() and execute it

```
class C {
  public static void main(String[] args) {
     System.out.println("C-main");
  }
}
public class A {
  static void m1() {
     System.out.println("m1");
  }
 public static void main(String[] args) {
     System.out.println("A-main");
  }
}
class B {
  static void m2() {
     System.out.println("M2");
  }
}
```

- Even if we have multiple main() in class, JVM will execute always from first main()
- Even in a single prg, we can have multiple classes to be public

2. java.util.Collection interface
    - toArray(Object[]) - Old version
    - toArray(IntFunction) - used to generate a new array from existing collection - new version

```
public class Main {
   public static void main(String[] args) {
          ArrayList<String> l1=new ArrayList<>();
          l1.add("one");
          l1.add("two");

          String s1[]=l1.stream().toArray(s->new String[s]); //old version
          System.out.println(Arrays.toString(s1));  //[one,two]
```

```
            String s2[]=l1.toArray(s11->new String[s11]); //new version
            System.out.println(Arrays.toString(s2));  //[one,two]

            String s3[]=l1.toArray(String[]::new); //method reference to constructor
            System.out.println(Arrays.toString(s3));  //[one,two]
    }
}
```

3. String methods

1. isBlank() - check whether String is blank or not
2. lines() - based on \n to extract list of strings. Fronm multi line string if we want to extract substring of each line
3. strip() - remove leading and trailing space
   stripLeading()
   stripTrailing()
strip() removes unicode char, whereas trim() dosent remove
4. repeat() - repeat string for particular times
5. transform() - transform string to different set
6. formatted() - format the string

```
public class Main {
    public static void main(String[] args) {
            String s1=" Welcome ";
            System.out.println(s1.isBlank()); //false
            String s2="   ";
            System.out.println(s2.isBlank()); //true
            System.out.println(s2.isEmpty()); //false

            String s3=" one \n"
                                +"two \n"
                                +"three \n";
            s3.lines().forEach(System.out::println);
            System.out.println(s1.repeat(4));

            System.out.println("  ".isBlank());  //true
            System.out.println(" L R ".strip().replace(" ","@")); //L@R
            System.out.println(" L R ".stripLeading().replace(" ","@")); //L@R@
            System.out.println(" L R ".stripTrailing().replace(" ","@")); //@L@R

            String s4="test string\u205F";
            String s5=s4.strip();
            System.out.println(s5+"*****");
            String t1=s4.trim();
            System.out.println(t1+"*****");

            System.out.println("HELLO".transform(s->s.substring(2))); //LLO

            System.out.println("My name is %s, and age is %d".formatted("Ram",25));
    }
}
```

4. Optional class
   - Optional.isEmpty() - to check whether optional is empty or not

```
public class Main {
   public static void main(String[] args) {
        Optional op1=Optional.of("Ram");
        op1.ifPresent(s->System.out.println(s));
        op1=op1.empty();
        System.out.println(op1.isEmpty());
        op1.ifPresent(s->System.out.println(s));
   }
}
```

5. Files read and write string
   - writeString(), readString(), isSameFile()

```
public class Main {
   public static void main(String[] args) {
        var path="C:\\Training\\data.txt";
        try {
                Files.writeString(Path.of(path), "welcome", StandardOpenOption.APPEND);
                String data=Files.readString(Path.of(path));
                System.out.println(data);
        }
        catch(Exception e) {
                System.out.println(e);
        }
   }
}
```

6. Predicate.not()

```
public class Main {
        public static boolean isEven(Integer n) {
                return n%2==0;
        }
   public static void main(String[] args) {
      List<Integer> l1=List.of(3,4,5,67,89,88);

      Predicate<Integer> p=n->n%2==0;
      l1.stream().filter(p).forEach(System.out::println);  //4,88

      l1.stream().filter(p.negate()).forEach(System.out::println); //3,5,67,89

      l1.stream().filter(Predicate.not(Main::isEven)).forEach(System.out::println);
   }    //3,5,67,89
}
```

7. TimeUnit conversion
   - used to convert DAY, HOURS, MINUTES, SECONDS

```
public class Main {
   public static void main(String[] args) {
     TimeUnit t1=TimeUnit.DAYS;
     System.out.println(t1.convert(Duration.ofHours(24)));  //1
```

```
        TimeUnit t2=TimeUnit.MINUTES;
        System.out.println(t2.convert(Duration.ofHours(60))); //3600

        long timeInSec=300L;
        System.out.println("Time: "+timeInSec+
                        " sec to min: "+t2.convert(timeInSec,TimeUnit.SECONDS)); //5

        long hrs=96;
        long days=TimeUnit.DAYS.convert(hrs,TimeUnit.HOURS);
        long min=TimeUnit.MINUTES.convert(days,TimeUnit.DAYS)
                        System.out.println(hrs+" "+days+" "+min);
    }
}
```

8. Pattern recognizing method
   -Predicate asMatchPredicate() - Pattern class - used to check given String matches the pattern, used in streams at time of filtering

```
public class Main {
    public static void main(String[] args) {
        Pattern p=Pattern.compile([file://d%7b4%7d-/d%7b2%7d-/d%7b2%7d]\\d{4}-\\d{2}-\\d{2});
        Predicate<String> p1=p.asMatchPredicate();

        Stream.of("2024-01-23","11-10-2021","2000-01-20","abcd-ef-gh")
            .filter(p1)
            .forEach(System.out::println); //2024-01-23   2000-01-20

        List<String> l1=List.of("Ram","Sam","Raj","Tam","Jam");
        Predicate<String> p2=Pattern.compile("^R.*$").asPredicate(); // internally uses Matcher.find()
        Predicate<String> p3=Pattern.compile("^R.*$").asMatchPredicate(); //internally uses Matcher.matches()

        List<String> l2=l1.stream().filter(p2).collect(Collectors.toList());
        System.out.println(l2); //[Ram,Raj]
        List<String> l3=l1.stream().filter(p3).collect(Collectors.toList());
        System.out.println(l3); //[Ram,Raj]
    }
}
```

Java 12 - Mar 2019 - Non LTS release

1. Compact Number format
   - used for representing the numbers in a human readable format

```
public class Main {
    public static void main(String[] args) {
        NumberFormat nf=NumberFormat.getCompactNumberInstance(Locale.US, Style.SHORT);
        String s1=nf.format(1000);
        System.out.println(s1);  //1K

        String s2=nf.format(100000000);
        System.out.println(s2); //100M

        String s3=nf.format(10000000000000L);
        System.out.println(s3); //10T

        String s4=nf.format(9999);
        System.out.println(s4); //10K  display nearest value
    }
}
```

2. Switch Expression
   - new form of switch statement, normally we cant return value from Switch stmt, instead we can go for switch expr
   - break is not required
   - If we want to return value we can use "yield" keyword

Syntax: label -> logic

```
public class Main {
  public static void main(String[] args) {
    String value=task("MONDAY");
    System.out.println(value);
  }

  public static String task(String day) {
        String msg=null;
        msg=switch(day) {
        case "MONDAY" -> {
                System.out.println("Lazy day");
                yield "Monday is lazyday";
        }
        case "TUESDAY" -> "Tuesday";
        case "WEDNESDAY" -> "Wednesday";
        case "THURSDAY" -> "Thursday";
        case "FRIDAY" -> "Friday";
        case "SATURDAY" -> "Saturday";
        default -> "Wrong day";
        };
        return msg;
  }
}
```

-We can have multiple case in single line

```
public class Main {
  public static void main(String[] args) {
    String value=task("MONDAY");
    System.out.println(value);
  }

  public static String task(String day) {
        String msg=null;
        msg=switch(day) {
        case "MONDAY" -> {
                System.out.println("Lazy day");
                yield "Monday is lazyday";
        }
```

```
            case "TUESDAY" -> "Tuesday";
            case "WEDNESDAY" -> "Wednesday";
            case "THURSDAY" -> "Thursday";
            case "FRIDAY","SATURDAY","SUNDAY" -> "Weekend starts";
            default -> "Wrong day";
            };
            return msg;
    }
}
```

3. Teeing Collector - Streams API
    - new static method teeing() in java.util.stream.Collectors interface, which takes two independent collectors and merge their results

Syntax: static Collector teeing(Collector c1,Collector c2,BiFunction b);

```
public class Main {
    public static void main(String[] args) {
        double d=Stream.of(1,2,3,4,5,6,7)
            .collect(Collectors.teeing(Collectors.summingDouble(i->i),
                        Collectors.counting(), (sum,count)->sum/count));
        System.out.println(d); //4.0

        List<Employee> elist=new ArrayList<>();
        elist.add(new Employee(1,"Ram",100));
        elist.add(new Employee(2,"Sam",200));
        elist.add(new Employee(3,"Raj",300));
        elist.add(new Employee(4,"Jam",400));

       HashMap<String,Employee> map= elist.stream().collect(Collectors.teeing(
                    Collectors.maxBy(Comparator.comparing(Employee::getSalary)),
                    Collectors.minBy(Comparator.comparing(Employee::getSalary)),
                     (e1,e2)->{
                            HashMap<String,Employee> hm=new HashMap<>();
                            hm.put("MAX", e1.get());
                            hm.put("MIN", e2.get());
                            return hm;
                     }));
       System.out.println(map);

    }

}
```

4. String methods

1. indent(n) - adjust the indention of each line of string based on argument
if n>0 - insert space at beginning of each line
if n<0 - remove space at beginning of each line
if n=0 - no change

2. transform(Function f)
3. Optional<String> describeConstable() - give description of String instance
4. resolveConstantDesc(MethodHandles m, LookUp l) - return description

```
public class Main {
   public static void main(String[] args) {
      String s1="Welcome \nto Java";
      System.out.println(s1.indent(0));
      System.out.println(s1.indent(3));

      String s2="Java";
      Object s3=s2.transform(val->new StringBuilder(val).reverse().toString());
      System.out.println(s3); //avaJ

      Optional<String> op=s2.describeConstable();
      System.out.println(op);

      String s4=s2.resolveConstantDesc(MethodHandles.lookup());
      System.out.println(s4);
   }
}
```

5. File mismatch method
   - check file content is mismatch or not

Syntax: static long mismatch(Path p1,Path p2)

If there is no mismatch then it returns 1L, if there is a mismatch then it returns position of first mismatch in file

```
public class Main {
   public static void main(String[] args) throws IOException {
     Path p1=Files.createTempFile("file1", ".txt");
     Path p2=Files.createTempFile("file2", ".txt");

     Files.writeString(p1, "hello");
     Files.writeString(p2, "hello");

     long mismatch=Files.mismatch(p1, p2);
     if(mismatch > 1L)
           System.out.println("Mismatch occured at "+mismatch);
     else
           System.out.println("No mismatch");

     Path p3=Files.createTempFile("file3", ".txt");
     Files.writeString(p3, "helloworld");
     long mismatch1=Files.mismatch(p1, p3);
     if(mismatch1 > 1L)
           System.out.println("Mismatch occured at "+mismatch1);
     else
           System.out.println("No mismatch");
   }
}
```

Java 13 - Sep 2019 - non LTS support

1. Switch expression - still preview feature

2. Text blocks - is a multi line literal, used to avoid the need of escape sequences and automatically formats the string in a predicatable way
   - denoted using """
   - u cannot have text block in single line

```
public class Main {
   public static void main(String[] args) {
           //old version
       String s1="hello welcome, \n"
                     +" to java \"bean\" .... \n"
                     +" welcome \n"
                     +" hello world \n";
       System.out.println(s1);

       //Text blocks
       String s2="""
                         hello world
                         to Java "bean"..
                         welcome
                         hello world
                       """;
       System.out.println(s2);

       String s3="""
                       Line 1: %s
                       Line 2: %s
                       Line 3:
                       Line 4:
                     """.formatted("Ram","Sam");
       System.out.println(s3);

       //String s4="""abcdxyz""";  //error

   }
}
```

3. String methods for text blocks
1. formatted(Object...args)
2. stripIndent() - used to remove white space char from the beginning and end of every line in text block
3. translateEscapes() - return string whose value is a string with escapes sequences translated as if in a string literal

```
public class Main {
   public static void main(String[] args) {
           String s1="""
                               Name: %s
                               Phone: %d
                               Salary: $%.2f
                               """.formatted("Ram",12345678,2000.6635);
           System.out.println(s1);

           String s2="<html>    \n"+
                    "\t<body>\t\t  \n"+
                                "\t\t<p>Hello</p>  \t \n"+
                    "\t</body>\n"+
                                "</html>";
           System.out.println(s2.replace(" ","*"));
           System.out.println(s2.stripIndent().replace(" ","*"));

           String s3="Hi\t\nHello \" /u0022 Pam\r";
           System.out.println(s3);
           System.out.println(s3.translateEscapes());
   }
}
```

Subject: RE: Reg: Day 3 Java Features Notes and Handson

Classification: **Internal**

Handson

Given:

```
public class PhoneBook {

    private static final HashMap<String, String> PHONE_NUMBERS = new HashMap<String, String>() {
        {
            put("Jos de Vos", "016/161616");
            put("An de Toekan", "016/161617");
            put("Kris de Vis", "016/161618");
        }
    };

    private HashMap<String, String> phoneBookEntries = PHONE_NUMBERS;

    PhoneBook() { }

    public HashMap<String, String> getPhoneBookEntries() {
        return phoneBookEntries;
    }

     public Optional<String> findPhoneNumberByName(String name){
        return null;
    }

    public Optional<String> findNameByPhoneNumber(String phoneNumber){
        return null;
    }

    @Override
    public String toString() {
        System.out.println("Hello from PhoneBook's toString method");
        return "PhoneBook{" +
                "phoneBookEntries=" + phoneBookEntries +
                '}';
    }
}


import java.util.Optional;
import java.util.stream.Stream;

public class PhoneBookCrawler {

   private PhoneBook phoneBook;

   public PhoneBookCrawler(PhoneBook phoneBook) {
      this.phoneBook = phoneBook;
   }

   public String findPhoneNumberByNameAndPunishIfNothingFound(String name){
      return null;
   }

   public String findPhoneNumberByNameAndPrintPhoneBookIfNothingFound(String name){
      return null;
   }

   public String findPhoneNumberByNameOrNameByPhoneNumber(String name, String phoneNumber){
      return null;
   }

   public PhoneBook2 getPhoneBook(){
      return phoneBook;
   }

}
```

1. Implement findPhoneNumberByName in PhoneBook class that returns an optional. (No streams)

2. Implement findPhoneNumberByNameAndPunishIfNothingFound in PhoneBookCrawler that uses the implementation from exercise 1

3. Implement findPhoneNumberByNameAndPrintPhoneBookIfNothingFound in PhoneBookCrawler that uses the implementation from exercise 1

4. Implement findNameByPhoneNumber in PhoneBook class that returns an optional. Implement findPhoneNumberByNameOrNameByPhoneNumber in PhoneBookCrawler class. First search the phone book by name. If that returns nothing search the phone book by phone number. If that still returns nothing return the phone number of Jos de Vos.

Java 9
1. JPMS
Transitive dependency
   1. If we are using module explicitly requires to use other module which is not practice, so we have to use transitive dependency

Circular(cyclic) dependency

- Circular dependency is not allowed in modular prg

Aggregator modules
- It is a modules which does not provide any functionality of its own, used to bundle together bunch of modules. It contains single module-info.java

Module Resolution process
- JVM will resolve the required modules in the modulepath before execution. First it will resolve root modules (starting of the module which contains class with main())

--show-module-resolution - show the module resolution process

Observables modules
- modules which are observed by JVM at runtime

2. JLink (Java Linker)
- To create our own customize JRE and execute on platform without Java installation, so Java is suitable for portable devices, microservices, iot devices. The size of Java env is reduced using JLink

```
C:\Training\example>javac --module-source-path src --module-path . -d out -m myModuleA

C:\Training\example>java --module-path out;. --module myModuleA/myPack.M1
Error: Could not find or load main class myPack.M1 in module myModuleA

C:\Training\example>java --module-path out;. --module myModuleA/myPack1.M1
Modular Programming

C:\Training\example>java --module-path out --show-resolution-process --module myModuleA/myPack1.M1
Unrecognized option: --show-resolution-process
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.

C:\Training\example>java --module-path out --show-module-resolution --module myModuleA/myPack1.M1
root myModuleA file:///C:/Training/example/out/myModuleA/
myModuleA requires java.xml jrt:/java.xml
java.base binds java.desktop jrt:/java.desktop
java.base binds jdk.security.auth jrt:/jdk.security.auth
java.base binds java.management jrt:/java.management
java.base binds jdk.localedata jrt:/jdk.localedata
java.base binds java.logging jrt:/java.logging
java.base binds jdk.zipfs jrt:/jdk.zipfs
java.base binds jdk.random jrt:/jdk.random
java.base binds java.xml.crypto jrt:/java.xml.crypto
java.base binds jdk.crypto.mscapi jrt:/jdk.crypto.mscapi
java.base binds java.security.sasl jrt:/java.security.sasl
java.base binds java.naming jrt:/java.naming
java.base binds java.smartcardio jrt:/java.smartcardio
java.base binds jdk.security.jgss jrt:/jdk.security.jgss
java.base binds java.security.jgss jrt:/java.security.jgss
java.base binds jdk.crypto.ec jrt:/jdk.crypto.ec
java.base binds jdk.crypto.cryptoki jrt:/jdk.crypto.cryptoki
java.base binds jdk.jartool jrt:/jdk.jartool
java.base binds jdk.compiler jrt:/jdk.compiler
java.base binds jdk.jdeps jrt:/jdk.jdeps
java.base binds jdk.jlink jrt:/jdk.jlink
java.base binds jdk.javadoc jrt:/jdk.javadoc
java.base binds jdk.jpackage jrt:/jdk.jpackage
java.base binds jdk.charsets jrt:/jdk.charsets
jdk.jpackage requires java.desktop jrt:/java.desktop
jdk.jpackage requires jdk.jlink jrt:/jdk.jlink
jdk.javadoc requires java.xml jrt:/java.xml
jdk.javadoc requires java.compiler jrt:/java.compiler
jdk.javadoc requires jdk.compiler jrt:/jdk.compiler
jdk.jlink requires jdk.jdeps jrt:/jdk.jdeps
jdk.jlink requires jdk.internal.opt jrt:/jdk.internal.opt
jdk.jdeps requires java.compiler jrt:/java.compiler
jdk.jdeps requires jdk.compiler jrt:/jdk.compiler
jdk.compiler requires java.compiler jrt:/java.compiler
jdk.crypto.cryptoki requires jdk.crypto.ec jrt:/jdk.crypto.ec
java.security.jgss requires java.naming jrt:/java.naming
jdk.security.jgss requires java.logging jrt:/java.logging
jdk.security.jgss requires java.security.sasl jrt:/java.security.sasl
jdk.security.jgss requires java.security.jgss jrt:/java.security.jgss
java.naming requires java.security.sasl jrt:/java.security.sasl
java.security.sasl requires java.logging jrt:/java.logging
java.xml.crypto requires java.xml jrt:/java.xml
java.xml.crypto requires java.logging jrt:/java.logging
jdk.security.auth requires java.naming jrt:/java.naming
jdk.security.auth requires java.security.jgss jrt:/java.security.jgss
java.desktop requires java.datatransfer jrt:/java.datatransfer
java.desktop requires java.xml jrt:/java.xml
java.desktop requires java.prefs jrt:/java.prefs
java.prefs requires java.xml jrt:/java.xml
java.desktop binds jdk.unsupported.desktop jrt:/jdk.unsupported.desktop
java.desktop binds jdk.accessibility jrt:/jdk.accessibility
jdk.compiler binds jdk.javadoc jrt:/jdk.javadoc
java.management binds java.management.rmi jrt:/java.management.rmi
java.management binds jdk.management jrt:/jdk.management
java.management binds jdk.management.jfr jrt:/jdk.management.jfr
java.naming binds jdk.naming.dns jrt:/jdk.naming.dns
java.naming binds jdk.naming.rmi jrt:/jdk.naming.rmi
java.datatransfer binds java.desktop jrt:/java.desktop
java.compiler binds jdk.javadoc jrt:/jdk.javadoc
java.compiler binds jdk.compiler jrt:/jdk.compiler
jdk.naming.rmi requires java.rmi jrt:/java.rmi
jdk.naming.rmi requires java.naming jrt:/java.naming
jdk.naming.dns requires java.naming jrt:/java.naming
jdk.management.jfr requires jdk.jfr jrt:/jdk.jfr
jdk.management.jfr requires jdk.management jrt:/jdk.management
jdk.management.jfr requires java.management jrt:/java.management
jdk.management requires java.management jrt:/java.management
java.management.rmi requires java.rmi jrt:/java.rmi
java.management.rmi requires java.naming jrt:/java.naming
java.management.rmi requires java.management jrt:/java.management
jdk.accessibility requires java.desktop jrt:/java.desktop
jdk.unsupported.desktop requires java.desktop jrt:/java.desktop
java.rmi requires java.logging jrt:/java.logging
```

Modular Programming

```
C:\Training\example>jlink --module-path "C:\Softwares\openjdk-17.0.1_windows-x64_bin\jdk-17.0.1\jmods"/;out
--add-modules myModuleA --output image

C:\Training\example>cd image

C:\Training\example\image>cd bin

C:\Training\example\image\bin>java --module-path out --module myModuleA/myPack1.M1
Modular Programming

C:\Training\example\image\bin>java --list-modules
java.base@17.0.1
java.xml@17.0.1
myModuleA
```

JLink Launcher
 - we have created mininal execution env inside image directory, but to execute that image file we have to provide very long command which is not good, so we create own our launcher to launch the app

```
C:\Training\example>jlink --module-path "C:\Softwares\openjdk-17.0.1_windows-x64_bin\jdk-17.0.1\jmods"/;out
--add-modules myModuleA --compress=2 --launcher runapp=myModuleA/myPack1.M1 --output image1

C:\Training\example>cd image1/bin

C:\Training\example\image1\bin>runapp
Modular Programming
```

Java10
 - In Mar 2018 which is time based versioning

```
int a=10;
String s="hello";
```

1. Local variable type inference - var keyword
   - used to infer the datatype at runtime depending on the value
   - used only inside constructor, methods and loops, compound block

Rules
1. var cant declare without initial value, it should be initialized in same line
2. var can be declared in first line and initialized in second line
3. var cannot be initialized with null value without a type
4. var is not permitted in multiple variable declaration
5. var cannot be used to initialize an array
6. var cant be used in parameters, return type and instance fields
7. var is a reserved type name but not reserved keyword, so it can be used as an identifier except for class, interface and enum
8. Take care of using var with <>
9. we can use var in lambda expr, but dont mix var and non var parameters

```
interface MyInterface {
        void add(int a,int b);
}
```

```
/*class var {  //error
}
interface var{ //error
}
enum var{ //error
}*/

class Var {
        public void var() {
                var var="var";
        }
        public void Var() {
                Var var=new Var();
        }
}
public class Main {
   Main() {
        var s=10;   //int
   }
   /*Main(var name){ //error

   }*/
   {  //compound block
        var s1="hello";  //String
   }
   //var i=10;  //error
   //static var i2=10;  //error
   public static void main(String[] args) {
      var a=3.14f;  //float
      for(var i=0;i<5;i++) {

      }
      for(var j:args) {

      }
      //var a2; //error
      var a1=0;
      var s2
           =2.3; //double
      //var s3=null;  //error
      var s3="hello";
      s3=null;
      var s4=(String)null;
      var c=10;
      //c=null; //error

      int w1=10, w2=20;
      //var d1=20, d2=23; //error

      //var f[]= {1,2,3}; //error

      List<String> l1=new ArrayList<>();
      var l2=new ArrayList<>();  //ArrayList<Object>
      var l3=new ArrayList<String>();  //ArrayList<String>
```

```
        MyInterface m1=(var a3,var b3)->System.out.println(a3+b3); //correct
       // MyInterface m2=(var a3,int b3)->System.out.println(a3+b3); //error
    }

    /*public var getName() { //error
           return "Ram";
    }*/
}
```

2. API for creating unmodifiable collection
- List.copyOf() - immutable
- Set.copyOf() - immutable
- Map.copyOf() - immutable
- toUnmodifiableList(),toUnmodifiableSet(),toUnmodifiableMap() - unmodifiable

3. Optional class
- orElseThrow()

```
public class Main {

    public static void main(String[] args) throws Throwable {
      List<Integer> l1=List.of(1,2,3);
      List<Integer> l2=List.copyOf(l1);  //immutable
      System.out.println(l2); //[1,2,3]
      //l2.add(4);  //exception

      Stream<String> st=Stream.of("a","b","c","d");
      List<String> l3=st.collect(Collectors.toUnmodifiableList());
      Stream<String> st1=Stream.of("a","b","c","d");
      Set<String> s4=st1.collect(Collectors.toUnmodifiableSet());

      Optional op=Optional.empty();
      System.out.println(op); //Optional.empty
      System.out.println(op.orElse("Jack")); //Jack
      System.out.println(op.orElseGet(()->"Peter")); //Peter
      System.out.println(op.orElseThrow(NullPointerException::new));
    }


}
```

Subject: RE: Reg: Day 2 Java Features Notes and Handson

Classification: **Internal**

Handson
1. Create arthimetic operation in one module and do those operation in another modules

JDK 9 Features
1. Private and private static methods in interface
2. Try with resource enhancement

3. Stream API enhancement
4. Optional class enhancement
5. Diamond operator enhancement
6. @Safevarargs annotation
7. Factory methods to create immutable collection

8. JShell
    - tool to study Java at beginning stage
    - REPL - Read Evaluate Print and Loop

```
>jshell

jshell> 10+20
$1 ==> 30

jshell> 10 < 20
$2 ==> true

jshell>  to come out ctrl+D

jshell> System.out.println("hello");
hello
```

Declaring Variables

```
jshell> int i = 10;
i ==> 10

jshell> i
i ==> 10

jshell> int j=20;
j ==> 20

jshell> j
j ==> 20

jshell> String str="hello";
str ==> "hello"

jshell> str
str ==> "hello"

jshell> int k;
k ==> 0

jshell> k
k ==> 0

jshell> k=40;
k ==> 40

jshell> int j=30;
j ==> 30
```

```
jshell> j
j ==> 30

jshell> String j="hi";
j ==> "hi"

jshell> j
j ==> "hi"
```

Scratch variable

- Whenever we create an expression and we are not assigning it to something, then Jshell will do 2 thing, first it will calculate the value and store in a temporary variable $1,$2

```
jshell> 10+20
$15 ==> 30

jshell> 10<20
$16 ==> true

jshell> $15
$15 ==> 30

jshell> $16
$16 ==> true

jshell> int i=$15+20;
i ==> 50
```

JShell commands

```
jshell> /vars
|    String str = "hello"
|    int k = 40
|    String j = "hi"
|    int $15 = 30
|    boolean $16 = true
|    int i = 50
```

- which tells all the variables that we have created in session

jshell>/history - print all the commands that we have executed so far

jshell>/list - it also gives the java commands and not jshell commands in a numerical order

```
jshell> /list

   1 : System.out.println("hello");
   3 : i
   5 : j
   6 : String str="hello";
   7 : str
   8 : int k;
   9 : k
  10 : k=40;
  12 : j
  13 : String j="hi";
  14 : j
  15 : 10+20
  16 : 10<20
  17 : $15
  18 : $16
  19 : int i=$15+20;

jshell> /15
10+20
$20 ==> 30
```

- It pulls the command based on serial number and reexecutes it and creates a new scratch variable

```
jshell> /13
String j="hi";
j ==> "hi"

jshell> j="hello";
j ==> "hello"

jshell> /13
String j="hi";
j ==> "hi"

jshell> j
j ==> "hi"

jshell> /!   - rerun the previous command
j
j ==> "hi"
```

jshell> press ctrl+r - used to search the command based on each character

jshell>/reset - reset the state of the current session

```
jshell> /reset
|  Resetting state.

jshell> /vars  - It will be empty because reset will erase all in the current
                session
```

Methods

```
jshell> System.out.println("hello");
hello

jshell> String greeting() {
   ...>     return "hi";
   ...> }
|  created method greeting()

jshell> greeting()
$3 ==> "hi"
```

```
jshell> String greeting(String name){
   ...>     return "Hello "+name;
   ...> }
|  created method greeting(String)

jshell> greeting("Ram");
$5 ==> "Hello Ram"

jshell> String greeting() {
   ...>     return "hiiiii";
   ...> }
|  modified method greeting()

jshell> greeting()
$7 ==> "hiiiii"
```

Forward Reference

When we create a method we can refer to variables that dont exists but we cant invoke the method

```
jshell> int  i=10;
i ==> 10

jshell> i=j;
|  Error:
|  cannot find symbol
|    symbol:   variable j
|  i=j;
|    ^

jshell> int add(){
   ...>     return i+j;
   ...> }
|  created method add(), however, it cannot be invoked until variable j is declared

jshell> add()
|  attempted to call method add() which cannot be invoked until variable j is declared

jshell> int j;
j ==> 0

jshell> add();
$12 ==> 10

jshell> void sayHelloWorld(){
   ...>     sayHello();
   ...>     sayWorld();
   ...> }
|  created method sayHelloWorld(), however, it cannot be invoked until method sayHello(), and method
sayWorld() are declared

jshell> void sayHello(){
   ...>     System.out.println("Hello");
   ...> }
```

```
|  created method sayHello()

jshell> void sayWorld(){
   ...>     System.out.println("World");
   ...> }
|  created method sayWorld()

jshell> sayHelloWorld();
Hello
World
```

Exceptions

```
jshell> 1/0
|  Exception java.lang.ArithmeticException: / by zero
|        at (#17:1)

jshell> int divide(int a,int b){
   ...>     return a/b;
   ...> }
|  created method divide(int,int)

jshell> divide(10,2);
$19 ==> 5

jshell> divide(1,0);
|  Exception java.lang.ArithmeticException: / by zero
|        at divide (#18:2)
|        at (#20:1)

jshell> int divide(int a,int b) throws IOException {
   ...>     if(b==0){
   ...>         throw new IOException();
   ...>     }
   ...>     return a/b;
   ...> }
|  modified method divide(int,int)

jshell> divide(1,0);
|  Exception java.io.IOException
|        at divide (#21:3)
|        at (#22:1)


jshell>/imports - display inbuild packages
```

Creating class

```
jshell> class Person {
   ...>     public String name;
   ...> }
|  created class Person

jshell> Person p=new Person();
p ==> Person@29444d75
```

```
jshell> p.
equals(      getClass()    hashCode()    name         notify()      notifyAll()
toString()    wait(
jshell> p.name="Ram";
$25 ==> "Ram"

jshell> p.name
$26 ==> "Ram"

jshell> $26
$26 ==> "Ram"

jshell> class Person {
   ...>     String fname;
   ...>     String lname;
   ...> }
|  replaced class Person
|    update replaced variable p, reset to null

jshell> p.
equals(      fname        getClass()    hashCode()    lname        notify()
notifyAll()   toString()    wait(
jshell> private class abc{
   ...> }
|  created class abc

jshell> abstract class xyz {
   ...> }
|  created class xyz

jshell> new xyz();
|  Error:
|  xyz is abstract; cannot be instantiated
|  new xyz();
|  ^--------^
```

9. JPMS(Java Platform Module System)
 Modules also a group of packages + module-info.java

Until Java8 everything is jar filebased, we have certain problems
1. If JVM unable to identify required .class file then we get NoClassDefFoundError
2. Version conflict
3. No Security
4. Size is big

All the above problems are called jar hell or classpath hell, we solve it using modular programming
1. So in modular programming JVM will check in the beginning only that all dependent modukles are available or not, if anything is missing immediately JVM will say so and so modules are not present, so it avoid NoClassDefFoundError
2. Which u want to export only that package is available to outside world
3. which module is required only that module is used by JRE, so size is small

```
jshell> var x="hello";
```

```
x ==> "hello"

jshell> x.getClass().getModule();
$32 ==> module java.base

jshell> x.getClass().getPackage();
$33 ==> package java.lang
```

>java --list-modules - display all inbuilt modules

module-info.java - specify it is modular program, if we didnt specify module-info.java then it is considered as unnamed module
- we will specify which package you export and which module is required for ur module
- Unnamed module can access all exported packages from named modules

```
module sample {
  exports com.pack;
  requires moduleA;
}


public class Main {

        public static void main(String[] args) {
                BufferedImage b=new BufferedImage(2,1,1);
                System.out.println(b.getClass().getModule());
        }

}


module Sample {
        exports com.pack;
        requires java.desktop;
}
```

Qualified Exports
Sometimes a module want to export its packages to specific modules, not to all modules

Syntax: exports packagename to module1, module2...

Classification: **Internal**

Handson

1. Create Employee class with id, name, age, gender, department, yearofjoining, salary

2. Create list of employee values

```
List<Employee> employeeList = new ArrayList<Employee>();
```

```
employeeList.add(new Employee(111, "Jiya Brein", 32, "Female", "HR", 2011, 25000.0));
employeeList.add(new Employee(122, "Paul Niksui", 25, "Male", "Sales And Marketing", 2015, 13500.0));
employeeList.add(new Employee(133, "Martin Theron", 29, "Male", "Infrastructure", 2012, 18000.0));
employeeList.add(new Employee(144, "Murali Gowda", 28, "Male", "Product Development", 2014, 32500.0));
employeeList.add(new Employee(155, "Nima Roy", 27, "Female", "HR", 2013, 22700.0));
employeeList.add(new Employee(166, "Iqbal Hussain", 43, "Male", "Security And Transport", 2016, 10500.0));
employeeList.add(new Employee(177, "Manu Sharma", 35, "Male", "Account And Finance", 2010, 27000.0));
employeeList.add(new Employee(188, "Wang Liu", 31, "Male", "Product Development", 2015, 34500.0));
employeeList.add(new Employee(199, "Amelia Zoe", 24, "Female", "Sales And Marketing", 2016, 11500.0));
employeeList.add(new Employee(200, "Jaden Dough", 38, "Male", "Security And Transport", 2015, 11000.5));
employeeList.add(new Employee(211, "Jasna Kaur", 27, "Female", "Infrastructure", 2014, 15700.0));
employeeList.add(new Employee(222, "Nitin Joshi", 25, "Male", "Product Development", 2016, 28200.0));
employeeList.add(new Employee(233, "Jyothi Reddy", 27, "Female", "Account And Finance", 2013, 21300.0));
employeeList.add(new Employee(244, "Nicolus Den", 24, "Male", "Sales And Marketing", 2017, 10700.5));
employeeList.add(new Employee(255, "Ali Baig", 23, "Male", "Infrastructure", 2018, 12700.0));
employeeList.add(new Employee(266, "Sanvi Pandey", 26, "Female", "Product Development", 2015, 28900.0));
employeeList.add(new Employee(277, "Anuj Chettiar", 31, "Male", "Product Development", 2012, 35700.0));
```

3. Find the solution using Java streams api

a. Print the name of all departments in the organization?
b. What is the average age of male and female employees?
c. Get the details of highest paid employee in the organization?
d. Get the names of all employees who have joined after 2015?
e. How many male and female employees are there in the sales and marketing team?
f. What is the average salary of male and female employees?
g. What is the average salary and total salary of the whole organization?
h. Who is the oldest employee in the organization? What is his age and which department he belongs to?

Java 5
1. For each stmt
2. Generics

```
    List<String> l=new ArrayList<String>();
```

3. Autoboxing and Unboxing

```
void add(int a,int b){
  sop(a+b);
}
void add(int a,int b,int c){
  sop(a+b+c);
}
void add(int a,int b,int c,int d){
  sop(a+b+c+d);
}
```

4. var args

```
void add(int...a){
  sop(a+b);
}
void add(int...a,boolean...b){   //only one var args in method
}
void add(String s,float...f){ //var args should be always present as last arg
}


System.out.println();

public class System extends Object {
   static PrintStream out;
}


public class Main {
   public static void main(String[] args)
   {
      double d=Math.sqrt(16);
      System.out.println(d);
   }
}
```

5. static import - used to invoke static methods and static variables without classname.methodname or classname.variablename

```
import static java.lang.System.*;
import static java.lang.Math.*;

public class Main {
   public static void main(String[] args)
   {
      double d=sqrt(16);
      out.println(d);
   }
}
```

6. StringBuilder class - mutuable class but not synch or thread safe, it gives better performance

7. Assertions - used to check boolean condition at runtime, if condition is failure it raise an AssertionError exception

8. Annotations

Java 7
1. Generics
```
      List<String> l=new ArrayList<String>();  //JDK1.5
      List<String> l=new ArrayList<>();  //JDK1.7
```

2. Multi catch statement - we can define multiple exception in single catch using |

```
try {

}
catch(ArthimeticException | NullPointerException | ClassCastException e){ }
```

3. switch(exp)
     exp - int/byte/char/enum/float/String(JDK1.7)

4. try with resources - no need explicitly close the resources and the resources should be present inside try, it will take care by AutoCloseable interface

5. Underscore literal - used for representation purpose

```
int a=100000;
int a=1_00_000;  //correct
int a=10_000; //correct
int a=_10000;  //error
float f1=3.1_4f;  //correct
float f1=3._14f; //error
float f1=3_.14f; //error
float f1=3.14_f; //error
```

6. Binary literal

```
    int a=0b10;   sop(a); //2
    int a=0B100;  sop(a); //4
```

Java 8 - Mar 18th 2014
1. Stream API
2. Functional Interface - only one abstract methods
3. Lambda expr - used to enable functional prg in java - we can write functions that exists on its own
4. Method reference using ::
5. Date API
6. Optional class
7. Default and static method inside interface

Java 9 - Sep 21st 2017
1. Private and private static method inside interface
 - If several default methods have same common functionality, then there may be a chance of duplicate code, so we can define the duplicate code inside private methods and call that in default method when it is needed

Interface member access
1. Accessible from default and private method inside interface
 constant variable = yes
 abstract method = yes
 another default method = yes
 private method = yes
 static method = yes
 private static method = yes

2. Accessible from static method inside interface
 constant variable = yes
 abstract method = no
 another default method = no
 private method = no
 static method = yes
 private static method = yes

3. Accessible from instance method implementing interface
 constant variable = yes
 abstract method = yes
 another default method = yes
 private method = no

static method = yes
private static method = no

4. Accessible outside interface without implementing
    constant variable = yes
    abstract method = no
    another default method = no
    private method = no
    static method = yes
    private static method = no

```
interface MyInterface {
        //till JDK1.7
        /*public final static*/ int CONSTANT=0;
        /*public abstract*/ int abstractMethod();

        //From JDK1.8
        /*public*/ default int defaultMethod() {
                abstractMethod();
                privateMethod();
                staticMethod();
                privateStaticMethod();
                return CONSTANT;
        }
        /*public*/ static int staticMethod() {
                privateStaticMethod();
                return CONSTANT;
        }

        //From JDK1.9
        private int privateMethod() {
                abstractMethod();
                defaultMethod();
                staticMethod();
                privateStaticMethod();
                return CONSTANT;
        }
        private static int privateStaticMethod() {
                return CONSTANT;
        }
}
class Example implements MyInterface {

        @Override
        public int abstractMethod() {
                defaultMethod();
                MyInterface.staticMethod();
                return CONSTANT;
        }

}
class Example1 {
        public int instanceMethod() {
                MyInterface.staticMethod();
                return MyInterface.CONSTANT;
        }
}
public class Main {
   public static void main(String[] args)
   {

   }
}
```

2. try with resources enhancement
   - we can access the resources outside the try block

```
class MyResource implements AutoCloseable {
        MyResource() {
                System.out.println("Resource Creation");
        }
        public void doProcess() {
                System.out.println("Resource processing");
        }
        @Override
        public void close() throws Exception {
                System.out.println("Resource closing");
        }
}
public class Main {
        public static void java6() throws Exception {
                MyResource r=null;
                try {
                        r=new MyResource();
                        r.doProcess();
                }
                catch(Exception e) {
                        System.out.println("Exception caught");
                }
                finally {
                        if(r!=null)
                                r.close();
                }
        }
        public static void java7() {
                try(MyResource r=new MyResource();){
                        r.doProcess();
                }
                catch(Exception e) {
                        System.out.println("Exception caught");
                }
        }
        public static void java9() {
                MyResource r=new MyResource();
                try(r) {
                        r.doProcess();
                }
                catch(Exception e) {
                        System.out.println("Exception caught");
                }
        }
        public static void multiResource() {
                MyResource r1=new MyResource();
                MyResource r2=new MyResource();
                MyResource r3=new MyResource();
                MyResource r4=new MyResource();
                try(r1;r2;r3;r4) {
                        r1.doProcess();
                        r2.doProcess();
                        r3.doProcess();
                        r4.doProcess();
                }
                catch(Exception e) {
                        System.out.println("Exception caught");
                }
        }
   public static void main(String[] args) throws Exception
   {
      java6();
      java7();
      java9();
      multiResource();
   }
}
```

3. Factory methods to create unmodifiable collection

Collections.unModifiableXXX() - just read-only views of the original collection, we can perform modifying operation on original collection and those modification will be reflected in unmodifiable collection, but we can modify unmodifiable collection

Immmutable - 100% u cant change anything - Before JDK1.9
    Collections.emptyList()
    Collections.nCopies(int n, String s)

```
public class Main {

   public static void main(String[] args)
   {
      List<String> l1=new ArrayList<>();
      l1.add("one");
      System.out.println(l1); //[one]

      List l2=Collections.unmodifiableList(l1);
      System.out.println(l2); //[one]
     // l2.add("two");
      //System.out.println(l2); //exception

      l1.add("two");
      System.out.println(l1); //[one,two]
      System.out.println(l2); //[one,two]
```

```
        //Immutable list
        List<String> l3=Collections.unmodifiableList(new ArrayList<>(l1));
        l1.add("three");
        //l3.add("three");  //exception

        System.out.println(l1); //[one,two,three]
        System.out.println(l2); //[one,two,three]
        System.out.println(l3); //[one,two]
    }
}
```

From JDK1.9, to create immutable collection we have

```
List.of(T...t)
Set.of(T...t)
Map.of(K...k)

List l1=List.of();  //empty list
List<Integer> l2=List.of(1);
List<Integer> l3=List.of(1,2,3,4,5);

Set l1=Set.of();  //empty set
Set<Integer> l2=Set.of(1);
Set<Integer> l3=Set.of(1,2,3,4,5);

Map l1=Map.of();  //empty map
Map<Integer,String> l2=Map.of(1,"One");
Map<Integer,String> l3=Map.of(1,"One",2,"two,3,"three");
Map<Integer,String> l4=Map.ofEntries(entry(1,"one"),entry(2,"two"),entry(3,"three"));
```

4. Stream API enhancement

1. default methods called
   takeWhile(Predicate) - if predicate is not satisfied it will discard the rest of stream
   dropWhile(Predicate) - if predicate is not satisfied it will print the rest of stream

2. Stream.iterate() with 3 args with conditional check

```
Syntax: static Stream iterate(T seed,Predicate p,UnaryOperator u) //JDK1.9
        static Stream iterate(T seed,UnaryOperator u) //JDK1.8
```

3. Stream.ofNullable(T t) - to prevent NPE and avoid null checks in stream

```
public class Main {

    public static void main(String[] args)
    {
        Stream.of(2,4,6,8,9,10,12).takeWhile((n)->n%2==0)
            .forEach(System.out::println);  //2,4,6,8
        Stream.of(2,4,6,8,9,10,12).dropWhile((n)->n%2==0)
            .forEach(System.out::println); //9,10,12

        Stream.iterate(1, x->x<5, x->x+1).forEach(System.out::println);
        List l1=Stream.ofNullable(100).collect(Collectors.toList());
        System.out.println(l1); //[100]

        List l2=Stream.ofNullable(null).collect(Collectors.toList());
        System.out.println(l2); //[]

        List<String> l3=new ArrayList<>();
        l3.add("A");
        l3.add("B");
        l3.add(null);
        l3.add("C");
        l3.add("D");
        l3.add(null);

        List<String> l4=l3.stream().flatMap(o->Stream.ofNullable(o)).collect(Collectors.toList());
        System.out.println(l4); //[A,B,C,D]
    }
}
```

5. Diamond Operator Enhancement
- From JDK1.5, <> it is used to specify type of objects and applied on both sides, can be applied only to classes
- From JDK1.7, <> it is used to specify type of objects and applied on left sides, can be applied only to classes
- From JDK1.9, <> can be applied to classes as well as Anonymous inner class

```
interface MyInterface<T> {
        void add(T a,T b);
}

public class Main {

    public static void main(String[] args)
    {
        MyInterface<Integer> m=new MyInterface<Integer>() {
            public void add(Integer a1,Integer a2) {
                    System.out.println(a1+a2);
            }
        };
        m.add(10, 20);
    }
}
```

6. Optional class enhancement

1. ifPresentOrElse(Consumer, Runnable) - if the value is present in Optional it returns the value, if it is absent it return another action

2. stream() - to convert Optional to Stream

```
public class Main {

    public static void main(String[] args)
    {
     Optional<String> op1=Optional.of("Ram");
     op1.ifPresentOrElse(a->System.out.println("Name: "+a), ()->System.out.println("NoName"));

     Optional<String> op2=Optional.empty();
     op2.ifPresentOrElse(a->System.out.println("Name: "+a), ()->System.out.println("NoName"));

    List<Optional<String>> l1 = Arrays.asList(Optional.of("Ram"),Optional.of("Sam"),Optional.of("Raj"));
    l1.stream().filter(o->o.get().contains("R")).forEach(System.out::println);

    List<String> l2=l1.stream().filter(o->o.get().contains("R"))
                              .flatMap(Optional::stream).collect(Collectors.toList());
    System.out.println(l2); //[Ram,Raj]
   }
}
```

7. @Safevarargs annotation applicable for private methods

Thanks and Regards
T.Senthil Kumar
8870845595