Java New features

Syllabus

Java 8 :→
- Default Method in interface
- Static Method in interface
- Functional Interface
- Predicate Interface
- Function Interface
- Consumer Interface
- Method Reference
- Lambda Expression
- Stream API
- forEach method
- Type Annonation
- Repeating Annonation

Java 9 :→
- → try with resource
- → diamond syntax with anonymus inner class
- → java module system

Java 10 → Local variable type inference

Java 12-17 → switch expression
- → yield keyword
- → Text Block
- → Record
- → Sealed classes

Default Method in interfaces

→ Before Java 8, we can only make abstract method in interface but there is a problem in it, if a new method add in interface then all implementing class has compile time error.

→ To solve this problem, default method introduced in java 8. We can make default method by using default keyword. Every implementing class can use this default method implementation or if it wants, implementing class can override.

```
interface A
{
    public abstract void m1();
    default void m2()
    { Sopln("default m2"); }
}
```

```
class B implements A
{
p v m3(){Sopln("m3");}
p s v main(String[] args)
{ A a = new B();
a.m2();→ (default m2)
}
```

```
class C implements A
{
p v m2(){Sopln("C");}
p s v main(String[] args)
{ A a = new C();
  a.m2(); → (C)
}
}
```

→ There may arise diamond problem by using default method, if two interfaces have same method signature.

→ To solve diamond problem, implementing class can override that method; if it writes:-

Interface1.super.m1() ⇒ Interface1 m1() method

Interface2.super.m2() ⇒ Interface2 m2() method

Or it can provides its own implementation.

If it not override m1() than CE

```
interface A
{
    default void m1()
    {
        Sopln("A m1");
    }
}
```

```
interface B
{
    default void m1()
    {
        Sopln("B m1");
    }
}
```

```
class Test implements A,B
{
    public void m1()
    {
        Sopln("Test m1");
        // A.super.m1(); → A m1
        // B.super.m1(); → B m1
    }
    public static void main(String[] args)
    {
        B b = new Test();
        b.m1(); → Test m1
    }
}
```

Static Method in Interface

→ To add common method in interface, that can be directly called by interface name.

→ We make static method by using static keyword.

→ Static method can only called by, interface name not by implementing class object so there is no diamond problem.

```
interface A
{
    static void m1()
    {
        Sopln("A m1");
    }
}
```

```
interface B
{
    static void m1()
    {
        Sopln("B m1");
    }
}
```

```
class Test implements A,B
{
p s v main(String[] args)
{
    A.m1(); → A m1
    B.m1(); → B m1
    A a = new Test();
    a.m1(); → CE
}
}
```

Functional Interface

→ Interface which has exactly one abstract method, called functional interface.

→ It may have any no. of default & static method.

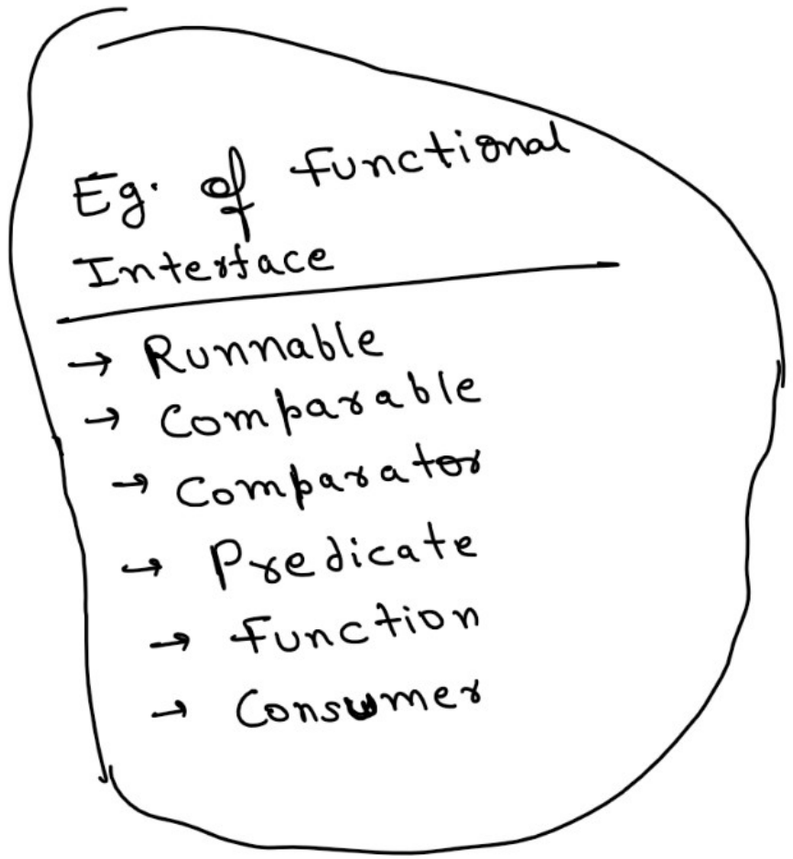
→ interface A
{
p abstract void m1();
}

→ interface B
{
p abstract void m1(); X
p abstract void m2();
}

→ @FunctionalInterface
interface A
{
p abstract void m1();
}

→ @FunctionalInterface
interface B → CE
{
p abstract void m1();
p abstract void m2();
}

```
@FunctionalInterface
interface A
{   public abstract void m1();
}
class Test implements A
{
    public void m1()
    {   Sopln("m1");
    }
    public static void main(String[] args)
    {   A a = new Test();
        a.m1();
    }
}
```



Predicate Interface

→ Predicate<T> is a functional interface.

→ It is present in java.util.function package.

→ It mainly used for condition checking.

```
interface Predicate<T>
{
    boolean test(T t);
}
```

→
```
import java.util.function.Predicate;
class Test implements Predicate<String>
{
    public boolean test(String s)
    {
        return s.length() %2 == 0;
    }
}
```

```
p s v main(String[] args)
{
  Predicate<String> p =
            new Test();
  Sopln( p.test("harsh"));
                ↳ false
}
```

Function Interface

→ Function<T, R> is a functional interface.

→ It is present in java.util.function package.

→ It mainly used for processing.

```
interface Function<T, R>
{
    R apply(T t);
}
```

→
```
import java.util.function.Function;
class Test implements Function
                    <String, Integer>
{
    public Integer apply(String s)
    {
        return s.length()-2;
    }
}
```

```
p s v main(String[] args)
{
Function<String, Integer> f =
                    new Test();
  Sopln(f.apply("harsh")); → 3
}
}
```

Consumer Interface

→ It is functional interface.

→ It is present in java.util.function package.

→ interface Consumer<T>
{
    void accept(T t);
}

→ import java.util.function.Consumer;
class Test implements Consumer<Integer>
{
    void accept(Integer n)
    {
        Sop ln(s);
    }

{
p s v main(String[] args)
{
Consumer<Integer> c =
        new Test();
  c.accept(10); → 10
}
}

Lambda Expression

→ It is used for reducing length of functional interface, implementing class code.

→ We can remove abstract method, modifiers, return type & method name.

```
p v add(int a, int b)        ⇒   (int a, int b) -> { Sopln(a+b); };
{ Sopln(a+b);
}
```

→ We can remove data type of parameters (a, b) -> { Sopln(a+b); };

→ If there is only one line then we can remove curly braces.

```
                                 (a, b) -> Sopln(a+b);
```

→ If that one stmt is return stmt then we have to remove return.

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

→ If there is only one parameter then we can remove parameter also.

```
p int square(int a)          ⇒   a -> a*a;
{ return a*a;
}
```

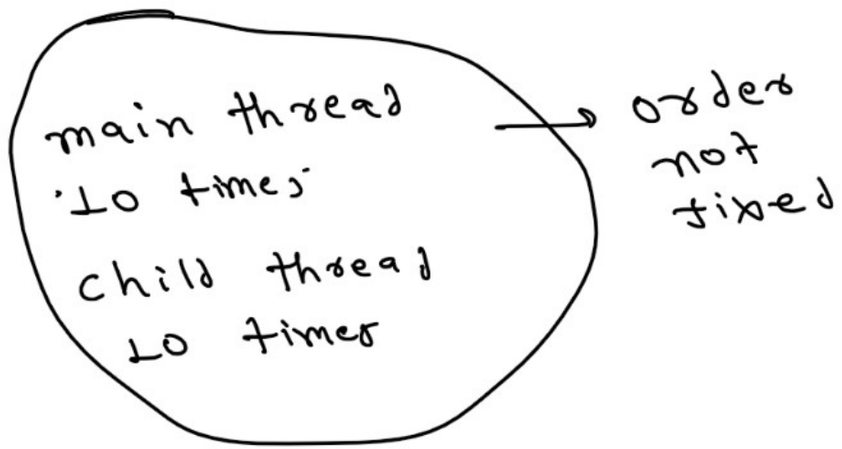
Lambda expression

```
interface Sum
{  int add(int a, int b);
}
interface Print
{  void print(String s);
}
interface Square
{   int sq(int a);
 }
```

```
class Test
{
p s v main(String[] args)
{
Sum s1= (a, b) -> a+b;
Print p1= s -> Sop ln(s);
Square s2 = a -> a*a;
Sop ln( s1. add(10,20)); → 30
 p1. print ("hello"); → hello
 Sop ln (s2. sq(10)); → 100
}
}
```

Lambda Expression for Runnable Interface

```
class Test
{
    p s v main(String[] args)
    {
        Runnable r = () -> { for(int i=0; i<10; i++)
                                Sopln("child thread");
                           };
        Thread t = new Thread(r);
        t.start();
        for(int i=0; i<10; i++)
            Sopln("main thread");
    }
}
```



Lambda Expression for Comparator Interface

```
import java.util.Comparator;
Class Employee
{
    int id;
    String name;
    Employee (int id, String name)
    {   this.id = id;
        this.name= name;
    }
    p String toString()
    {   return this.id + this.name;
    }
```

```
p s v main (String[] args)
{
Comparator<Employee> c =
(emp1,emp2) -> emp1.name.compareTo
                    (emp

TreeSet<Employee> t= new TreeSet<>(c);
t.add(new Employee( 1, "harsh"));
t.add (new Employee (2, "ravi"));
t.add (new Employee (2, "abc"));
Sopln (t); → [2 abc, 1 harsh, 2 ravi]

}
}
```

→ Lambda Expression can be used when functional interface used as method or constructor argument.

```
interface Sum
{ p int add (int a, int b);

}
interface Mul
{ p int mul (int a, int b);

}
p s v main (String[] args)
{
```

```
class Test
{ p s v m1 (int a, int b, Sum s, Mul m)
  {
     Sopln (s.add (a, b));
     Sopln (m.mul (a, b));
  }
  p s v main (String[] args)
  {
   m1 (10, 20, (a,b)->a+b,
               (a,b)->a*b);
  }
}
```

↳ 30
200

Method Reference

→ It is used to refer an existing method in the case of functional interface.

```
interface A
{
    p v m1();
}
class Abc
{
    p v m2()
    {
        Sopln("m2");
    }
}
```

(same) (same)

```
class Test
{
    p s v main(String[] args)
    {
        Abc obj = new Abc();
        A a = obj::m2;
        a.m2(); → (m2)
    }
}
```

If m2 is static

```
A a = Abc::m2;
a.m2(); → (m2)
```

→ Internally

```
class System
{   static PrintStream out;
}
```

and PrintStream class has println() method which has void return type.

& in Consumer interface void accept(T t)

So

```
class Test
{ p s v main (String[] args)
  {
    Consumer<String> c = System.out::println;
    c.accept("hello");
  }
}
```

→ hello

System.out → PrintStream object

println → method name

Stream API

→ Stream API is used to process elements of Collection. & Collection is group of object.

→ We can use Stream API to all collection classes like ArrayList, LinkedList, Vector, Stack, HashSet, LinkedHashSet, TreeSet, PriorityQueue:--

Steps :→

Step1 First convert collection into stream. by stream() method present in Stream Interface which is present in java.util.stream package.

public Stream<T> stream()

Step2 Process Stream.

→ public Stream<T> filter (Predicate<T> p)

→ public Stream<R> map (Function<T, R> f)

And some more methods are

→ public Stream<T> sorted (Comparator<T> c)

→ public Optional<T> min (Comparator<T> c)

→ public Optional<T> max (Comparator<T> c)

→ public long count()

Step3 And then again convert stream into collection by Stream, collect() method.

```
collect ( Collectors.toList()
          .toSet() )
```

```
import java.util.stream.Stream;
import java.util.stream.Collectors;
import java.util.List;
import java.util.ArrayList;

class Test
{
    p s v main(String[] args)
    {
        List<Integer> l = new ArrayList<>();
        l.add(10);
        l.add(5);
        l.add(20);

        Stream<Integer> s1 = l.stream();
        Stream<Integer> s2 =
                s1.filter(n -> n%2==0);
        List<Integer> l1 =
                s2.collect(Collectors.toList());
        Sopln(l1); → [10 20]

        Stream<Integer> s3 =
                s1.map(n -> n*n);
        List<Integer> l2 =
                s3.collect(Collectors.toList());
        Sopln(l2);      [100 25 400]
    }
}
```

→ We can also write in single line only like

```
List<Integer> l = new ArrayList<Integer>();
l.add(10);
l.add(20);
l.add(5);
List<Integer> l1 = l.stream().filter(n->n%2==0).map(n->n*2).
                   collect(Collectors.toList());
Sopln(l1); → [100 400]
List<Integer> l2 = l.stream().filter(n-> n-6>0).sorted
                   ((n1,n2)->-n1.compareTo(n2)).
                   collect(Collectors.toList());
Sopln(l2); → [20 10]
```

descending order

```
int minE = l.stream().map(n -> n*2).min((n1,n2) ->
                                  n1.compareTo(n2))
```

```
Sopln(minE); → 25
```

`.get();`

↓

to get particular object present in Optional<T>

```
int maxE = l.stream().map(n -> n*2).max(
                    (n1,n2) -> -n1.compareTo(n2)).get();
```

[40 20 10]
↓ min, ↓ max

```
Sopln(maxE); → 40
```

```
int count = (int) l.stream().filter(n -> n%2==0)
                                .count();
Sopln(count) → 2
```

forEach method

→ Can be used for Stream & Collection implementing classes

Void forEach (Consumer<T> c)

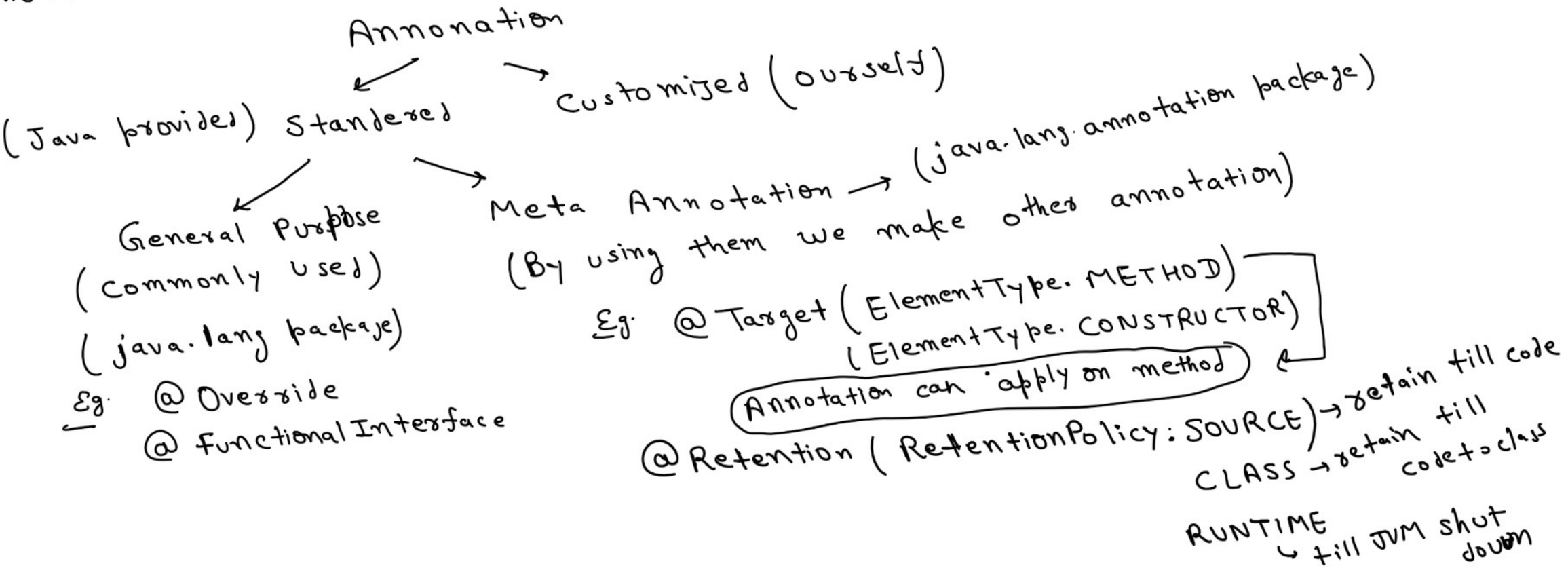
→
```
import java.util.List; import java.util.ArrayList;
class Test
{
    p s v main(String[] args)
    {
        List<Integer> l = new ArrayList<>();
        l.add(10);
        l.add(20);
        l.forEach(n -> System.out.println(n));   [10 20]
        l.forEach(System.out::println);          [10 20]
    }
}
```

Type Annonation

First we understand little bit about Annotation.

→ Annotation is used to tell compiler or JVM that how to treat this code.

→



We create our customized constructor by

```
@ interface AnnotationName
{
   datatype propertyname() default value;
}
```

default value; → optional

Type Annotation

→ Before Java 8 we can't apply annotation on type. In Java 8 we can apply annotation on any type used.

```
@Target(ElementType.TYPE_USE)
```

- can use before data type
- in throws clause
- method return type
- constructor call
- generic type

```
import java.util.annotation.Target;
import java.util.annotation.Retention;
import java.util.List; import java.util.ArrayList;
@Target(ElementType.TYPE_USE)
@Retention(RetentionPolicy.RUNTIME)

@interface TypeAnno
  {
  }

class Test
{
   p @TypeAnno int m1();
   p s v main() throws @TypeAnno Exception
   {
      @TypeAnno int x=10;
      @TypeAnno Test t=new Test();
      List<@TypeAnno Integer>
         l=new ArrayList<>();
   }
}
```

Repeating Annotation

→ Before Java 8, we can't apply one annotation at same place multiple time. but in java 8 we can apply by @Repeatable.

```
import java.util.annotation.Repeatable;
import java.util.annotation.Target;
import java.util.annotation.Retention;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Repeatable(AdminController.class)
@interface Admin
{
  String name();
}
```

```
@interface AdminController
{
  Admin[] value();
}
```

name must be value

```
@Admin(name="harsh")
@Admin(name="hello")
class Test
{
}
```

multiple

Try with Resource

→ Before java versions if we open any resource in try block then we have to close that resource in finally block. On try with resource we can write like try(resource) & if multiple resource try(resource1; resource2) then it will automatically close. so don't need to write finally block.

→
```
import java.io.FileWriter;
import java.io.FileReader;
import java.io.IOException;
class Test
{
```

```
p s v main(String[] args) throws IOException
{  FileReader f1 = new FileReader("abc.txt");
   try(FileWriter f = new FileWriter("abc.txt", true))
   {
        f.write(100); → 'd' write on abc.txt
        f.write(101); → 'e'
   }
   catch(IOException e) { Sopln(e); }

   try (f1)
   {  int i = f.read();
      while(i != -1)
      {  Sopln((char)i); → d e
         i = f.read();
      }
   }
   catch(IOException e) { Sopln(e); }
}
```

automatically close

abc.txt

diamond syntax with anonymous inner class

inner class → class inside class

anonymous → without name

Before Java 8 we can make anonymous inner class like →

→

```
class Test
{ p s v main(String[] args)
  {
    Thread t=new Thread()
    { p v run(){
        for(int i=0;i<10;i++)
          Sopln("child thread");
    };
    t.start();
    for(int i=0;i<10;i++)
      Sopln("main thread");
  }
}
```

It will create annonymous inner class in Test which is child of Thread

o/p →

main thread 10 times
child thread 10 times

→ Order not fixed

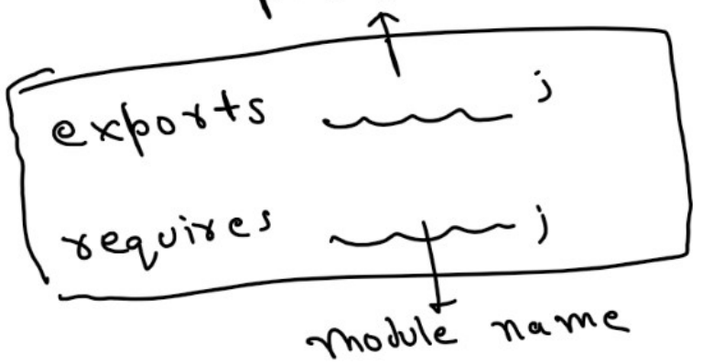
In Java 8 we can also make anonymyus inner class with diamond Syntax.

```
import java.util.List;
import java.util.ArrayList;
class Test
{
    p s v main(String[] args)
    {
        List<Integer> l=new ArrayList<>()
        {
            p boolean add(Integer i)
            {
                l.add(i*2);
            }
        };
        l.add(10);
        l.add(20);
        Sopln(l);→[20 40]
    }
}
```

Java Module System

→ Module is group of packages. We make modules to provide modularity, security & readability.

→ Every module has module-info.java where module can decide which module it requires & which packages it can export. by exports & requires.



Eg.

module 1
- package 1
  - Hello1.java
- package 2
  - Hello2.java
- module-info.java

module 2
- package 3
  - Hello3.java
- module-info.java

Assume Hello3.java want to use Hello1.java method then :→

Hello1.java

```
class Hello1
{  p v  m1()
    {  Sopln("Hello1 m1");
    }
}
```

Hello2.java

```
class Hello2
{
}
```

module-info.java

```
module module1
{  exports package1;
}
```

Hello3.java

```
import package1.Hello1;

Class Hello3
{
   p s v main(String[] args)
    {
       Hello1 h=new Hello1();
       h.m1();
    }
}
```

→ Hello1 m1

module-Info-java

```
module module2
{  requires module1;
}
```

Local Variable var Inference

from Java 10 we can use "var" keyword instead of writing full data type. like var x = 10;

→ Compiler looks at right side value 10 & decides x data type, int & can't change after that.

```
var x = 10;
x = "harsh"; → CE
```

→ Can't leave blank or assign null value;

```
var x; → CE
var y = null; → CE
```

→ can't use for functional interface type.

```
var p = n -> n % 2 == 0; → CE
```

→ can't use in method & constructor argument.

```
p int m1(var x) { }
```
→ CE

```
Test(var x) { }
```
→ CE

We can use var keyword for

→ local variable not for instance & static variable.

```
var x = 10;
```

→ inside loop

```
for(var i=0; i<10; i++)
{
    Sopln(i);
}
```

→ for collection

```
var list = new ArrayList<Integer>();
```

```
import java.util.ArrayList;
Class Test
{
    p s v main(String[] args)
    {
        var x=10;
        Sopln(x);                    → 10
        for(var i=0; i<10; i++)
            Sopln(i);                → 0 1 ... 9
        var l=new ArrayList<>();
        l.add(10);
        Sopln(l);                    → [10]
    }
}
```

Switch Expression & yield keyword

→ Traditionally switch is a statement not a expression. & after every case we have to write break; to save from fallthrough.

```
switch(x)
{   case 1: Sopln("1");
            break;
    case 2: Sopln("2")
            break;
    default: Sopln("3");
}
```

can use

byte, short, int, char

+

Byte, Short, Integer, Character, enum → java 1.5

+

String → java 1.7

+

any Object

In switch expression we can use

→ multiple label in one case.

→ arrow after case to return value.

→ no break required.

```
int x = 1;
String result = switch (x)
                {
                    case 1,2 -> "Monday";
                    case 3 -> "Tuesday";
                    default -> "Wednesday";
                };
Sopln(result);
```

→ Monday

If there are multiple lines in case & we have to return something, then we can use yield keyword.

```
int x = 1;
var result = switch(x)
{
    case 1,2 -> { Sopln("hello"); yield x; }
    case 3,4 -> 2;
    default -> 3;
};
Sopln(result);
```

"hello" → hello

result → 1

Text Block

→ TextBlock is a multiline string literal that improves readability & avoid using many escape characters.

Before

```
String str = "{ \n" +
    " \"name\" : \"Harsh\",\n" +
    " \"age\" : 20 \n" +
    "}";
```

After we can use

```
String str = """
    {
        "name" : "Harsh",
        "age" : 20
    }
    """ ;
```

Rules

① `String str = """` → can't write after this quote has to write in next line

```
harsh
""";
```

② After every line internally it automatically add \n character. To save from this we can add "\" character.

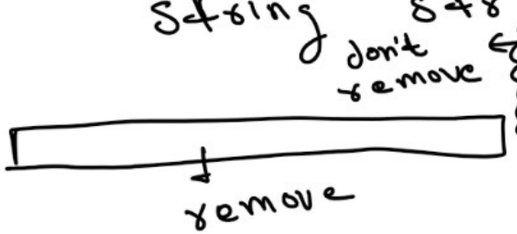
```
String str= """
            name: "harsh", \
            age: 10
""";
```

③ Will remove right most space internally. to save from this we can add "\s" character.

```
String str= """
            name: "harsh",  \s
""";
```

④ Will remove left most space internally from left most place.

```
String str= """
            name: "harsh",
            age: 20
""";
```



Record

→ It is special data carrier class.

→ If we normally make that class then we have to make constructor, getter methods, toString() method, equals() method & hashCode() method to reduce this length of code, we can make record.

Eg. If a class Test object carries property id, name so traditionally we have to make class like i-

```
class Test
{   int x;
    String y;
    Test(int x, String y)
    {   this.x = x;
        this.y = y;
    }
```

```
→ p String toString()
  { return id + name;
  }
→ p int getId()
  { return this.id;
  }
→ p String getName()
  { return this.name;
  }
→ p boolean equals(Object obj)
  {   if( this == obj)
          return true;
      if( obj == null)
          return false;
      if( this.getClass() != obj.getClass())
          return false;
```

don't need to remember complete code for exam

```
      Test t = (Test) obj;
      return this.id == t.id &&
             this.name.equals(t.name)
  }
→ p int hashCode()
  { return id;
  }
}
```

→ Now we can write this complete code by

```
record Test1(int id, String name)
{
}
```

```
class Main
{
    p s v main(String[] args)
    {
        Test t = new Test(1, "harsh");
        Sopln(t.getId());          → 1
        Sopln(t.getName());        → harsh

        Test1 t1 = new Test1(2, "rahul");
        Sopln(t1.Id());            → 2
        Sopln(t1.Name());          → rahul
    }
}
```

Sealed class

→ If any class wants to restrict which classes can inherit it, we can use "sealed" keyword & "permits" keyword.

```
Sealed class A permits B, C
{
}

class B extends A
{
}
```

→ Only B & C are allowed to inherit A

→ And child class must have 3 keyword either

- Sealed → only allowed classes can inherit B (indirectly inherit A)
- non-sealed → anyone can inherit B
- final → noone can inherit B

```
sealed class A permits B, C, D
{  p v m1() { Sopln ("m1"); }
}
sealed class B extends A permits E
{  p v m1() { Sopln ("B m1"); }

}
non-sealed class C extends A
{  p v m1() { Sopln("C m1"); }

}
final class D extends A
{  p v m1() { Sopln ("D m1"); }
}
```

→ Now E can extends B.
So indirectly it extend A.

→ Any class can extend C.

→ No class can extend D.