1 HOUR

①

Pg no. {1, 5, 11, 15, 16, 18, 21}

22 pages.

→ JAVA

MyApp.java → javac MyApp.java → MyApp.class

source file → compiler → Byte code

primitive Data types
→ will not contain null values

→ Primitive Data Types: (overflows) ..

overflow ? what comes after ~~is it~~ min (max values)

Type →	contains	default	size	Range (+ overflows) ✓
boolean	true or false	false	1 bit	NA
char	unicode character	\u0000	(2) 16 bits (2B)	\u0000 to \uFFFF (65535)
byte	signed integer	0	(1) 8 bits (1B)	-2^7 -128 to 127 2^7-1
short	signed integer	0	(2) 16 bits (2B)	-2^{15} -32768 to 32767 $2^{15}-1$
int (switch)	signed integer	0	(4) 32 bits (3B)	-2^{31} to $2^{31}-1$
long (l)	signed integer (100L)	0	(8) 64 bits (4B)	-2^{63} to $2^{63}-1$
float (f)	IEEE 754 f.p. (100f)	0.0	(4) 32 bits (3B)	1.4 e^{-45} to 3.4 e^{+38}
double	IEEE 754 f.p. (100d)	0.0	(8) 64 bits (4B)	4.9 e^{-324} to 1.8 e^{+308}

bcbsil/fd

char c = 'A' or char c = 65;

→ switch (n) {
int, short, char, byte, string
(boolean, float, long, double not supported)
boolean
Decimal not allowed.

```
switch (n) {
    case 1:
        break;
    case 2:
        break;
    default:
        break;
}
```

Public enum Days {SUN, MON, TUE}
Days d1, d2;
d1 = Days.wed

→ while:

```
int i=1;
while( i<=5){
    system.out.println("Hello");
    i++;
}
```

do while

```
int i=1;
do {
    system.out.println("Hello");
    i++;
}while (i<=5);
```

→ for loop

```
for(int i = 1; i<=5; i++){
    system.out.println("Hello");
}
```

foreach

```
int[] primes = new int[] {2,3,5,7};
for (int n: primes){
    system.out.println(n);
}
```

→ continue : used to skip current iteration

```
for(int i=0; i<10; i++) {
    if (i==3) {
        continue;
    }
    system.out.println(i);
}
```

skipped

whenever 'continue' is executed, it skips remaining statements in the block

Output:
0
1
2
4
5
6
7
8
9

what about nested loops:
(inside)
only the nested loop breaks/continues.
It doesn't impact outer loop.

→ break: break the loop

```
for(int i=0; i<10; i++){
    if (i==3) { // if (i>=3)
        break;
    }
    system.out.println(i);
}
```

Breaks the loop

Output:
0
1
2

```
for(int i=0; i<3; i++){
    for (int j=0; j<3; j++){
        if(i==1) {
            break/continue;
        }
        system.out.println(i);
    }
    system.out.println(i);
}
```

(only the inner for loop is impacted.)

4.1 : class and object

// same package → import statement not reqd.

objectdemo.java

```
class Calc {
    int num1;
    int num2;
    int result;
    public void perform() {
        result = num1 + num2;
    }
}
public class Objectdemo {
    public static void main(String[] args) {
        Calc obj = new Calc();
        obj.num1 = 3;
        obj.num2 = 5;
        obj.perform();
        System.out.println(obj.result);
    }
}
```

toString method

4.2 : Constructor

4.3 A.java :

```
class A {
    int i; //0
    float f; //0.0
    public A() {
        i = 5;
        f = 5.5f;
    }
    public A(int k) {
        i = k;
        f = 5.5f;
    }
}
```

default constructor

test.java

```
public class test {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new (6);
    }
}
```

Notes
- static variables are same for all objects
- Non static variables are different for different object
- cannot use non-static variable in static block.

4.4: this keyword

A.java

```
Public class A {
    int num1;
    int num2;
    Public A (int num1, int num2) {
        this.num1 = num1;
        this.num2 = num2;
    }
}
```

4.5: Method overloading
Constructor overloading

* class loader memory
* JVM memory
  stack, heap, etc.

4.6: static
value will be same for all objects

// Note: we can have multiple static { } blocks in a class. ① static { id=1; } ② static { id=2; } . value of id will be overwritten to 2

Emp.java

```
class Emp {
    int id;
    int salary;
    static int ceo;
    static {                    // when you load a class
        ceo = "larry";          → Initialize static members
        sysout("in static");
    }
    Public Emp() {              // when you create an object
        id = 1;
        salary = 5000;          → Initialize non-static members
        sysout("in constructor");
    }
    public void show() {
        sysout("id " + id + " salary " + salary + " " + ceo);
    }
}
```

StaticDemo.java

```
Public class staticDemo {
    Pub. static void main(...) {
        Emp emp1 = new Emp();
        Emp emp2 = new Emp();
        emp1.id = 1;
        emp1.salary = 3000;
        emp2.id = 2;
        emp2.salary = 4000;
        Emp.ceo = "xyz";
        emp1.show();
        emp2.show();
    }
}
// Emp.ceo = "Larry";
```

output:
```
in static
in constructor
in constructor
1 3000 xyz
2 4000 xyz
```

output:
~~in static~~
1 3000 ~~xyz~~
2 4000 xyz

Note:
- 'static' block is executed only once & is used to initialize static variables. constructor is used to initialize non-static variables.
- class is loaded only once during execution, whereas constructor is executed everytime you create an object

—contd..

4.7: Inner class

Outer.java

→ Outer.class

```
class Outer {
    int a;
    public void show(){
    }
```

→ Outer$Inner.class

```
    class Inner {
        public void display() {
        }
    }
}
```

member class

InnerDemo.java

```
public class InnerDemo {
    public static void main() {
        Outer obj = new Outer();
        obj.show();
        Outer.Inner obj1 =
                obj.new Inner();
        obj1.display();
    }
}
```

// Outer.Inner obj1 = (new Outer()).new Inner();

Outer.java

```
Class Outer {
    static int a;
    public static void show(){
    }
```

class Outer$Inner.class

```
    static class Inner {
        public void display() {
        }
    }
}
```

static class

InnerDemo.java

```
public class InnerDemo {
    public static void main () {
        Outer obj = new Outer();
        obj.show();
        Outer.Inner obj1 =
            new Outer.Inner();
        obj1.display();
    }
    //Outer.Inner.display()
}
// No need to create object for static class
```

Note: A class can have → Member class;
Static class
Anonymous class

~~it is not required for static inner class.~~

5.1 Array 1D | 2D | Jagged Array

→ `int[] num = new int[4];`
`num[0] = 0;`
`num[1] = 1;`
`num[2] = 2;`
`num[3] = 3;`
`num[4] = 4;` → ERROR: out of bound exception

(or)

`int[] num = {0,1,2,3,4}`

→ `student s1 = new student();`
`student s2 = new student();`
`student s3 = new student();`
`student[] s = {s1,s2,s3};`

→ `int[] a = {1,2,3,4};`
`int[] b = {2,4,6,8};`
`int[] c = {5,6,7,8};`
`int[][] d = {a,b,c};`

or

`int[][] d = {`
`{1,2,3,4},`
`{2,4,6,8},`
`{5,6,7,8}`
`};`

d[][]

row \ column	0	1	2	3
0	1	2	3	4
1	2	4	6	8
2	5	6	7	8

`d[0][2] = 3`

```
for(int i=0; i<3; i++) {
    for(int j=0; j<4; j++) {
        ...
    }
}
```

JAGGED ARRAY

→ `int[][] d = {`
`{1,2,3,4},`
`{2,4,6},`
`{5,6,7,8}`
`}`

d[][]

	0	1	2	3
0	1	2	3	4
1	2	4	6	
2	5	6	7	8

```
for(int i=0; i<d.length; i++) {
    for(int j=0; j<d[i].length; j++) {
        System.out.print(" " + d[i][j]);
    }
    System.out.println();
}
```

Output:
1 2 3 4
2 4 6
5 6 7 8

`d.length = 3` = no. of rows
`d[0].length = 4`
`d[1].length = 3`
`d[2].length = 4`

5.2: Enhanced for loop [for each loop]

→
```
int val[]={1,2,3,4}
for(int i : val)
    sysout(i);
}
```

1D

output:

1
2
3
4

→
```
int a[]={1,2,3,4};
int b[]={2,4,6,8};
int c[]={5,6,7,8}
```

d

	0	1	2	3
0	1	2	3	4
1	2	4	6	
2	5	6	7	8

2D

```
int[][] d = {
            {1,2,3,4},   → first element is an array
            {2,4,6},
            {5,6,7,8}
           };

for(int[] i ; d){
    for(int val : i){
        System.out.print(" " + i);
    }
    System.out.println();
}
```

5.3: varargs

Calc.java

```
Public class Calc{
    public int add(int ... n)// {4,5,6,7,8}
    {
        int sum=0;
        for(int i : n){
            sum+=i;
        }
        return sum;
    }
}
```

→ int[] n={4,5,6,7,8}

int...n
in between n space
NO space

```
public class VarargsDemo{
    public static void main(String[] args){
        Calc obj=new Calc();
        sysout2(obj.add(4,5,6,7,8));
    }
}
```

6.1 Inheritance

// Dog IS-A animal
// Bathroom HAS-A tub

// IS-A, HAS-A

```
class Calculator{
    public int add(int i, int j){
        return (i+j);
    }
}

class CalcAdv extends Calculator{
    public int sub(int i, int j){
        return (i-j);
    }
}

Class CalcVeryAdv extends CalcAdv{
    public int mul(int i, int j){
        return (i*j);
    }
}
```

single-level Inheritance

Single-level Inheritance

multi-level Inheritance

```
Public class InheritanceDemo{
    public static void main(String[] args){
        CalcVeryAdv c1 = new CalcVeryAdv();
        int res1 = c1.add(1,2);
        int res2 = c1.sub(3,2);
        int res3 = c1.mul(5,6);
    }
}
```

calc (calc = new CalcVeryAdv(); can access all methods in child classes)

NOTES:

- Java ~~supports~~ Single-level, multi-level inheritance ✓ ✓
- Java do not support multiple ~~level~~ inheritance ✗
- ~~Two~~ relationships in inheritance → IS-A; HAS-A
  - IS-A: A class extends another class, we have IS-A relationship.
    Ex: CalcAdv extends Calc ⇒ IS-A relationship
    Ex: Dog is an Animal → Dog extends Animal
  - HAS-A: A class reading object of another class, we have HAS-A relationship
    Ex: "InheritanceDemo" class has-a relationship with "CalcVeryAdv"
    Ex: Bathroom HAS-A tub

6.2 : super method

A)

```
class A {
    public A() {
        sysout2("In A");
    }
}

class B extends A {
    public B() {
        sysout2("In B");
    }
}
```

① . public class superdemo {
    Pub st. void main() {
        A a = new A();
    }
}

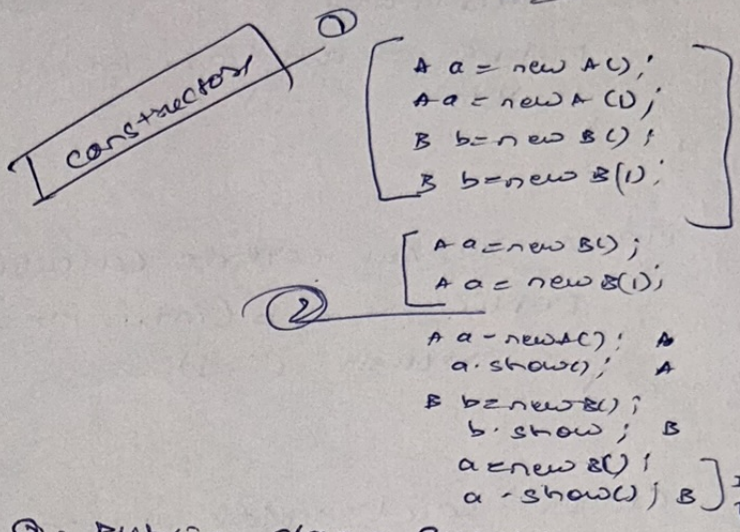
Output:
In A

② . public class superdemo {
    Pub. st. void main() {
        B b = new B();
    }
}

Output:
In A
In B

→ when we create object of subclass, it calls super/parent class constructor first & then calls constructor of subclass

SUPER



B)

```
class A
    public A(){
        sysout2("In A");
    }
    public A(int i){
        sysout2("In A int");
    }
}
class B extends A {
    public B(){
        //super();
        sysout2("In B");
    }
    public B(int i){
        //super();
        sysout2("In B int");
    }
}
```

① . public class superdemo {
    Pub st. void main() {
        B b = new B();
    }
}

outputs:
In A
In B

② . public class superdemo {
    Pub st void main() {
        B b = new B(3);
    }
}

outputs:
In A
In B int

Notes:
- every derived class constructor has "super()" method by default

c). class A
```
    public A(){
        sysout2("In A");
    }
    public A(int i){
        sysout2("In A int");
    }
}

class B extends A{
    public B(){
        super(); → calls default constructor of superclass
        sysout2("In B");
    }
    public B(int i){
        super(i); → calls parameterized constructor of superclass
        sysout2("In B int");
    }
}
```

① . public class superdemo{
```
    pub st void main(){
        B b = new B(5);
    }
}
```
outputs:
```
In A int
In B int
```

Q). How to dynamically call constructors at run-time?

Base class

Note:

- B b = new B() ⇒ In A
  In B

- If you want to call parameterized constructor in base class, use a random parameter in public B(){..}
```
super(5);
```
ie
```
public B(){
    super(5);
    sysout2("In B");
}
```
Now,
```
B b = new B() ⇒ In A int
                In B
```

6.3: Why Java does not support multiple inheritance
```
class A {
    show();
}

class B{
    show();
}

//Assume java supports multiple inh.
class C extends A,B {
}
```
```
main(){
    C obj = new C();
    obj.show();
}
```

Note:
- Here, java doesn't know which "show()" to call.
- It is called "ambiguity" problem
- Java solves ambiguity problem by removing multiple inheritance

6.4 Method overriding | super keyword

a).
```
class A {
    public void show() {
        sysout2("in A");
    }
}
class B extends A {
}
```

b).
```
class A {
    public void show() {
        sysout2("in A");
    }
}
class B extends A {
    public void show() {
        sysout2("in B");
    }
}
```

c).
```
class A {
    public void show() {
        sysout2("in A");
    }
}
class B extends A {
    public void show123() {
        sysout2("in B");
    }
}
```

①
```
public class OverridingDemo {
    pub. st void main(String[] args) {
        B obj = new B();
        obj.show();
    }
}
```
Output:
in A

①
```
public class OverridingDemo {
    pub st void main(String[] args) {
        B obj = new B();
        obj.show();
    }
}
```
Output:
in B

(#) Note:
- class B method overrides method show() of class A : method overriding
- child class overrides parent class method

①
```
B obj = new B();
obj.show();
```
Output:
in A

Note:
→ use @Override annotation to override parent class method to avoid compile time error due to typo

ie
```
class B extends A {
    @Override
    public void show() {
        sysout2("in B");
    }
```

(*)
```
// A a = new A();
   a.show(); → in A
// A a = new B();
   a.show(); → in B
// B b = new B();
   b.show(); → in B
// B b = new A(); → C.T ERROR
```

```
// A a = new A();
   a.show(); → in A
   a = new B();
   a.show(); → in B
```
DMD → RTP

d). to call both overriding & overridden method

```
class A {
    public void show() {
        ② sysout2("in A");
    }
}

class B extends A {
    public void show() {
        ② super.show();
        ③ sysout2("in B");
    }
}
```

```
① public class OverridingDemo {
    pub st void main() {
        B obj = new B();
        ① obj.show();
    }
}
```

Output:

in A
in B

e).

```
class A {
    int i;
    public void show() {
        sysout2("in A");
    }
}

class B extends A {
    int i;
    public void show() {
        super.i = 8;
        super.show();
        sysout2("in B");
    }
}
```

① .

```
B obj = new B();
obj.show();
```

"super.i" used to access parent class object

6.5) Dynamic method Dispatch

a).
```
class A {
    public void show() {
        sysout2 (" in A");
    }
}

class B extends A {
    public void show() {
        sysout2 (" in B");
    }
}

~~class C extends A {~~
    ~~public void show() {~~
```

① public class OverridingDemo{
```
Pub st void main() {
    A obj1 = new B();
    obj1.show();
}
}
```
(A obj1 → reference; new B() → implementation)

outputs

in B

others:

①. B obj1 = new B(); ⇒ in B
obj1.show();

②. A obj = new A(); ⇒ in A
obj1.show();

b).
```
class A {
    public void show(){
        sysout2 (" in A");
    }
}
class B extends A {
    public void show() {
        sysout (" in B");
    }
    public void config(){
        sysout(" config");
    }
}
class C extends A
    public void show(){
        sysout2 (" in C");
    }
}
```

① public class OverridingDemo{
```
Pub st void main() {
    A obj1 = new B();
    obj1.show();
    obj1.config();   // ERROR
}
}
```
- This will not work, because class A does not have method config()

-

② public class OverridingDemo{
```
Pub st void main(){
    A obj1 = new B();
    obj1.show();

    obj1 = new C();
    obj1.show();
}
}
```
output:

in B
in C

[Polymorphism
Encapsulation
Abstraction.
wrapping (Boxing)]

Defn:

c). **Two types of Polymorphism**

- Compile Time Polymorphism (static polymorphism) → method overloading
- Run Time polymorphism (Dynamic polymorphism) → method overriding

- Run-time polymorphism

```
Public class OverridingDemo{
    // compile time & run time
    Public static void main(String [] args){
        A obj  = new B(); // Runtime polymorphism
        obj.show();
        obj = new C();
        obj.show()  // Dynamic method dispatch
    }
}
```

Output:
in B
in C

This method overriding is called Run-time polymorphism

Notes;

- A obj = new B() ⇒ These two are linked during runtime & not during compile time
- since, we assign object during runtime; which show() method to call is also decided during runtime
- when we change the object, it calls a different method. This concept is called Dynamic method dispatch. Java uses Run time polymorphism to achieve DMD.

- compile-time polymorphism

~~This is achieved by function overloading~~
- Polymorphism is resolved during compile time Ex: method overloading

```
class calc {
    int add (int i, int j){
        return ( i+j);
    }
    int add (int i, int j, int k){
        return (i+j+k);
    }
}
Public class test {
    Pub st void main(String[] args){
        ~~sysout~~(
            calc obj=new calc();
            Sysout(obj.add(1,2)); // method overloading
            sysout(obj.add(1,2,3); // method overloading
        }
}
```

6.6) Encapsulation

// Encapsulation → Binding data with methods. only way to access data is using methods

```
class Student {
    private int rollNO;
    private String name;
    //Getters & setters
    public void setRollNo(int r){
        rollno = r;
    }
    public int getRollNo(){
        return rollNO;
    }
}
public class EncapsulationDemo{
    pub st void main (String[] args){
        Student s1 = new Student();
        s1.setRollNO(2);
        Sysout2( s1.getRollNO());
    }
}
```

Output:
2

Abstraction?
Hiding functionality implementation details
Ex: TV remote

6.7) Wrapper class | AutoBoxing

// primitive data types → int, float, double
// wrapper class → Integer, Float
(Hibernate uses only wrapper class)

concept is from 'c' programming
Data is stored in primitive data format.

```
public class WrapperDemo{
    public static void main(){
        int i=5; //Primitive datatype
        Integer ii = new Integer(i); // Boxing or wrapping
        int j = ii.intValue(); // unboxing or unwrapping
        Integer value = 9; // AutoBoxing
        int k = value;  // Auto-UnBoxing
        String str = "123";
        int n = Integer.parseInt(str);
    }
}
```

i → variable
ii → object

6.8) Abstract keyword

- Abstract class is created so that somebody can extend it in future
- you cannot instantiate (create object) of an abstract class
- Abstract method must always be in abstract class
- Abstract class can have non abstract method
- A class which extends Abstract class must compulsorily override parent abstract method & other undefined methods, other it gives an error and extended class also becomes abstract class).

```
abstract class Animal {  // Abstract class
    public abstract void eat();  → must be implemented in derived class
    public void walk() {
        sysout("walk");
    }                  → No need to implemat as it is already defined in parent class
}

class Man extends Animal {  // concrete class
    public void eat() {
    }                  → Non-abstract class need not be implemented (Ex: All animals can breathe)
    [ public void walk() {
        sysout("man walk");
      } ]  → optional and valid
}

Public class AbstractDemo {
    public static void main(String[] args) {
        Animal obj = new Man();
    }
}
```

→ creating 'Animal' object is meaningless
→ creating 'Man' object makes sense

6.9) Why do we need Abstract class. ↑

- we don't want anyone to create object of abstract class
- we can use one method (instead of several methods) which can accept all subclass objects

```
class Printer {
    public void show(Number i) {  → Number is an abstract class
        sysout(i);                → one method is created instead of show(Integer i) show(Double i) show(Float i)...
    }
}
Public class AbstractDemo {
    public static void main(String[] args) {
        Printer obj = new Printer();
        obj.show(1);   → Integer extends Number
        obj.show(1.2f); → Float extends Number
        obj.show(5.5); → Double extends Number
    }
}
```

- Abstract class can have final methods
- Abstract class can have constructors & static methods also

Solid Example:

- Let's say we have a class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, etc. Since the animal sound differs from one to another, there is no point to implement this method in parent class. Because every child must override this method to give its own implementation details like Lion will "Roar", Dog will "woof", etc.

So, we know that all animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus making this abstract would be the good choice. In abstract class, we force all child classes to implement this method.

- Abstract class is like a template. You have to extend it & build it before you can use it.
- A Non-Abstract classes are called concrete classes

Properties of Abstract class:

① We cannot create instance (objects) of an abstract class. But we can have references

```
abstract class Base {
    abstract void fun();
}
class Derived extends Base {
    void fun() { System.out("..."); }
}
```

```
class main {
    PSV main(String[] args) {
        Base b = new Derived();
        b.fun();
    }
}
```

output: ...

② Abstract class can have constructor. It is called when an instance of inherited (extended) class is created

```
abstract class Base {
    Base() { System.out("Base constructor called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out("Derived constructor called"); }
    void fun() { System.out("Derived fun() called"); }
}
class main() {
    PSV main() {
        Derived d = new Derived();
    }
}
```

constructor

Base b = new Derived(); what happens?

output: Base constructor called
Derived " "

(3). We can have abstract class w/o any abstract method. This allows us to create classes that cannot be instantiated but can only be inherited (extended)

```
abstract class Base {                    can't create object. Must be extended to call fun() method
    void fun() { System.out.println("Base fun() called"); }
}
class Derived extends Base { }                    can create object
class Main {
    public static void main() {
        Derived d = new Derived();
        d.fun();
    }
}
```

Non-abstract method

Base base = new Derived();
base.fun(); → what happens? error or System.out.???

output:

Base fun() called

(4). Abstract classes can have final methods (methods that cannot be overridden). (pto)

```
abstract class Base {
    final void fun() { System.out.println("fun() called"); }
}
class Derived extends Base { }
class Main {
    public static void main() {
        Base b = new Derived();
        b.fun();
    }
}
```

Final method

output: fun() called

(5). Abstract classes cannot be instantiated.

```
abstract class Test {
    public static void main(String[] args) {
        Test t = new Test();
    }
}
```

✗

output:

Compile time error. Test is abstract cannot be instantiated

6.10) Final keyword

- we can use final keyword for variables, methods or classes

① . variable

```
final int i = 10;   // CONSTANT
i = 20;  → ERROR
```

② . class:

```
final class A {
    public void show() {
        sysout("A");
    }
}

class B extends A {  → ERROR
}
```

⇒ cannot be inherited (extended)

③ method:

```
class A {
    public final void show() { sysout("A"); }
}

class B extends A {
    public void show() { sysout("B"); } → ERROR
}
```

→ cannot be overridden

Note:

Abstract classes and interfaces are both used for abstraction in programming, but they serve different purposes. Abstract classes provide a base class for other classes, allowing for code reuse and shared functionality, while Interfaces define contracts that classes must implement

When to use:

(a). Abstract classes: when you want to share common code and behavior among a group of related classes

(b). Interfaces: when you want to define a contract that multiple classes can adhere to, regardless of their relationship to each other.

7.1) Interfaces

```
class Writer{
    public void write(){ (or)
    }
}

class Pen extends Writer{
    @Override
    public void write(){
        System.out.println("I'm a Pen");
    }
}

class Pencil extends Writer{
    @Override
    public void write(){
        System.out.println("I'm a Pencil");
    }
}

class Kit{
    public void doSomething(Writer P){
        P.write();
    }
}

public class AbstractDemo{
    psvm(){
        Kit k = new Kit();
        Writer P = new Pen();
        Writer PC = new Pencil();
        k.doSomething(P);
        k.doSomething(PC);
    }
}
```

// loose coupling

```
abstract class Writer{
    public abstract void write();
}
```

(FA)

```
interface Writer{
    void write();
}

class Pen implements Writer{
    @Override
    public void write(){
        System.out.println("I'm a Pen");
    }
}

class Pencil implements Writer{
    @Override
    public void write(){
        System.out.println("I'm a Pencil");
    }
}
```

passing methods/functions as arguments using lambda fn's.

```
k.doSomething(()->"I'm Pen"));
k.doSomething(()->"I'm Pen"));
```

& Remove all implementations.

output:

```
I'm a Pen
I'm a Pencil
```

```
Kit kit = new Kit();
Writer writer = new Pen();
kit.doSomething(writer);
writer = new Pencil();
kit.doSomething(writer);
```

output: same as above

→ Interface → 100% abstraction ; Abstract classes → 0 to 100% abstraction
→ All methods in 'interface' are public abstract
→ Interface cannot be instantiated (objects cannot be created)
→ we cannot extend multiple classes (ie multiple inheritance)
but we can implement multiple interfaces
→ Always prefer interface

```
class Pen extends Writer1, Writer2{
    ... // wrong
}
```

```
class Pen implements Writer1, Writer2
}
    ... // correct
}
```

overcomes problem of not supporting multiple inheritance

Interfaces,

- used to achieve total abstraction
- overcomes problem of multiple inheritance
- used to achieve loose coupling
- Interfaces are used to achieve abstraction ~~III~~ to Abstract classes. ~~Exist~~ Abstract classes may contain non-final variables, whereas variables in interfaces are ~~final, public & static~~ check

```
interface A {
    final int a = 10;
    void display()
}
```

→

```
JDK ≥ 8 supports method definition
:
void display() {
    sysout(" --- ");
}
:
```

7.3) Anonymous Inner class

```
class A {
  Public void show{
    sysout (" A ");
  }
}
Class B extends A {
  Public void show(){
    sysout (" B ");
  }
}
Public class AnonymousEx {
  Psr main() {
    A obj = new B();
    obj.show();
  }
}
```

Output:

B

Here, only purpose of class B is to override class A method "show()". Then we can remove class B using an. classes.

⇒

```
class A {
  Public void show() {
    sysout (" A ");
  }
}
Public class Anonymous Ex {
  Psr main() {
    A obj = new A()
    { specify implementation
      Public void show() {
        sysout (" B ");
      }
    };
    obj.show();
  }
}
```

(specify object & then)

output:

B

7.4) Anonymous Inner class with Interface

```
interface Abc {
    void show();
}

class Implementer implements Abc {
    public void show() {
        sysout("anything");
    }
}

Public class InterfaceDemo {
    Psr main() {
        Abc obj = new Implementer();
        obj.show();
    }
}
```

Output:

anything

```
interface Abc {
    void show();
}

public class InterfaceDemo {
    Psr main() {
        Abc obj = new Abc()
        {
            public void show() {
                sysout("anything");
            }
        };
        obj.show();
    }
}
```

Output:

anything

7.5). Functional Interface / Lambda Expression

Types of Interface:

1. Normal Interface → Interface having ≥1 method
2. Single abstract method (SAM) interface → interface having only one method ie. (1 interface)
3. marker interface → doesn't have any method
   Ex) 'Serializable' interface
   Why do we need marker interface? Refer serialization

From JDK 1.8 & above SAM interface is called Functional Interface

@FunctionalInterface → user can use this annotation just to ensure that there is always one method

```
interface Abc {
    void show();
}
```

```
interface Abc {
    void show();
}

Public class InterfaceDemo{
  Psv main(){
    Abc obj=new Abc()
        {
          Public void show(){
            sysout("Hello");
          }
        };
    obj.show();
  }
}
```

⇒

```
interface Abc{
    void show();
}

Public class InterfaceDemo{
  Psv main(){
    Abc obj = () → sysout("Hello");
    obj.show();
  }
```

Output:

Hello

Output:

Hello.

Steps:

```
Abc obj = new Abc()
    {
      Public void show(){
          sysout("Hello");
      }
    };
```

⇒

```
Abc obj = () →
    { // removed : if only one stmt.
      sysout("Hello");
    }
```

⇓

```
Abc obj= () → sysout("Hello");
```

This stmt. belongs to show method.

7.6) Default method in Interface

// Abstract class → declare & define
// Interface → declare → JDK ≤ 1.7
→ declare & define → JDK ≥ 1.8

```
@Functional Interface
interface Demo{                    only one
    void abc(); // abstract method
    default void show(){
        sysout("in show");
    } // you can have any number of
      default methods
}

class DemoImpl implements Demo{
    public void abc(){
        sysout("in abc");
    }
    public void show(){
        // default method can be overridden
        sysout("in new show");
    }
}
```

→ It is not mandatory to implement default methods

```
public class InterfaceDemo{
  Psv main(){
    Demo obj=new DemoImpl();
    obj.abc();
    obj.show();
```

Output:

in abc
in new show

```
interface MyInterface {
    void existingMethod(String str); // Already existing Public and abstract method. All implementation classes must implement this method

    default void newMethod() {  // This is default method. So we need not implement this method in the implementation classes. (User can implement it required -> optional)
        System.out.println("newly added default method");
    }

    static void anotherNewMethod() {  // static method in interface is similar to default method except that we cannot override them in implementation classes.
        System.out.println("newly added static method");
    }
}

public class Example implements MyInterface {
    // implementing abstract method
    public void existingMethod(String str) {
        System.out.println("String is: " + str);
    }

    psvm main() {
        Example obj = new Example();
        // MyInterface -> will this work?
        // calling default method of interface
        obj.newMethod();

        // calling static method of interface
        MyInterface.anotherNewMethod();

        // calling abstract method of interface
        obj.existingMethod("Java 8 is easy");
    }
}
```

Need to override

can override

cannot override

Output:

```
Newly added default method
Newly added static method
String is : Java 8 is easy
```

- In earlier JDK, it was mandatory to implement all methods in an interface
- With default methods, users can/cannot implement, similar to abstract class

Abstract class v/s Interfaces

- Purpose of interface → provide full abstraction
  Purpose of abstract class → provide partial abstraction
- ✓ Using default methods, user can add additional features in the interface w/o affecting end user classes
- ✓ abstract class can have constructors, interface doesn't
- ✓ Interfaces can have concrete methods (methods with body) just like abstract classes in JDK ≥ 1.8
- ✓ multiple inheritance problem can occur when we have 2 interfaces with default methods of same signature

```
interface myInterface1{
    default void newMethod(){
        System.out.println("myInterface1");
    }
    void existingMethod(String str);
}

interface myInterface2 {
    default void newMethod(){
        System.out.println("myInterface2");
    }
    void disp(String str);
}

public class Example implements myInterface1, myInterface2 {
    //implementing abstract methods
    public void existingMethod(String str){
        System.out.println("String is: "+str);
    }

    public void disp(String str){
        System.out.println("String is: "+str);
    }

    //Implementation of duplicate default method
    public void newMethod(){
        System.out.println("Implementation of default method");
    }

    Psv main(){
        Example obj = new Example();
        //calling default method of interface
        obj.newMethod();
    }
}
```

Output:

Implementation of default method

```
interface X {
    public void myMethod();
}
interface Y {
    public void myMethod();
}
class JavaExample implements X, Y {
    public void myMethod(){
        sysout("Hello");
    }
    PSV main() {
        JavaExample obj = new JavaExample();
        obj.myMethod();
    }
}
```

Output:

Hello

- Why multiple inheritance is not supported in Java?

```
class A {
    void msg() { sysout("Hello");}
}
class B {
    void msg(){ sysout("Welcome");}
}
class C extends A, B { // suppose if it were
    PSV main (String[] args){
        C obj = new C();
        obj.msg(); // Now which msg() method would be invoked?
    }
}
```

Compile Time Error:

7.7) Multiple inheritance issue with interface

(a)
```
@FunctionalInterface
interface Demo {
    int num=100; // constant ie final int num=100
    void abc();
    default void show(){
        sysout("in Demo show");
    }
}

interface MyDemo{
    default void show(){
        sysout("in MyDemo show");
    }
}

class DemoImpl implements Demo,MyDemo{
    public void abc(){
        sysout("in abc");
    }
    public void show(){
        sysout("in DemoImpl show");
    }
}
```

is this correct check once

(1)
```
public class InterfaceDemo{
    psv main(){
        Demo obj=new DemoImpl();
        obj.abc();
        obj.show();
    }
}
```

output:
```
in abc
~~in Demo show~~
in DemoImpl show
```

(b) How to call show() method of interface instead of our implementation.

In above code replace show() method in DemoImpl
ie.

```
public void show(){
    sysout("In DemoImpl show");
}
```

⇒

```
public void show(){
    Demo.super.show();
    // MyDemo.super.show();
}
```

output;
```
in abc
in Demo show
// in MyDemo show
```

Note:
- Note that we have used super keyword with interfaces.

7.8) Static method in interface

8.1) Java package

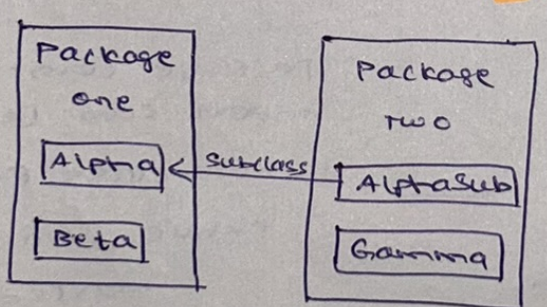
- package name should be mirror of domain name
- google.com → com.google → 100% unique package name

8.2) Access modifiers

- private & protected cannot be used with class
- Inner class can be private
- class and interface cannot be declared as private
- If a class has private constructor then you cannot create object of that class from outside the class

Scope of Access modifiers

modifier	class	package	subclass (same pkg)	subclass (diff. pkg.)	world
public	Y	Y	Y	Y	Y
protected	Y	Y	Y	Y	N
default	Y	Y	Y	N	N
private	Y	N	N	N	N



classes → public, default

members, methods, constructors of a class → private, public, protected, default

interface → public, default

members, methods of interface → by default is public if access modifier is not specified

visibility

Modifier	Alpha	Beta	AlphaSub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

Private constructor:

- Private constructor cannot be called from anywhere outside the class
- Private constructor can still get called from other constructors or from static methods in the same class

```
Public class Clock{
    Private long time=0;
    Private Clock(long time){
        this.time = time;
    }
    Public Clock(long time, long timeOffset){
        this(time);
        this.time += offset;
    }
    Pubic static Clock newClock(){
        return new Clock(System.currentTimeMillis());
    }
}
```

- Access modifier assigned to a class takes precedence over any access modifier assigned to fields, constructors & methods of that class

---

```
package com.telusko.test;
public class Student{
    int age=27;
    Public int rollNo=8;
    Private String name="Navin";
    Protected int marks=56;
}
```

```
package com.telusko;
import com.telusko.test;
Public class Engineer extends Student{
    Public void show(){
        // age=27
        marks=77;
        rollNo = 55;
        // name="Navin"
    }
} // Public, Protected
```

```
package com.telusko.test;
Public class DemoMain{
// Public class DemoMain extends Student{
    PSV main(){
        Student s = new Student();
        s.rollNo=9;
        s.marks=99;
        s.age=30;
    } // s.name → ERROR
}
// Public, Protected, default
```

age = 27
rollNo = 8
// name = "Navin"
marks = 56

```
package com.telusko;
import com.telusko.test.Student;
public class InnerDemo{
    PSV main(){
        Student s = new Student();
        s.rollNo=9;
        // age=30;
    }
} // Public
```

// age = 27
// rollNo = 8
// name = "Navin"
// marks = 56

9.1) try catch Finally

```
public class ExceptionDemo {
    psvm main() {
        try {
            int i=7;
            int j=0;
            int k = i/j;
        } catch (ArithmeticException e) {
            sysout(e);
        } finally {
            sysout("always executed");
        }
    }
}
```

// unchecked exception (Runtime exception)

// only subclasses of RuntimeException are unchecked. Rest are all checked exceptions

RTE

Exceptions: try{}catch()

- RTE → unchecked (subclasses of RTE)
- CTE → checked (others) throws or try-catch.

9.2) Multiple catch blocks

```
public class ExceptionDemo {
    psvm main {
        try {
            int k= 7/0;
        } catch (ArithmeticException | ArrayIndexOutOfBoundException e) {
            sysout(e);
        } catch (Exception e) {
            sysout(e); // should be last one always
        } finally {
            sysout("always executed");
        }
    }
}
```

// unchecked exception (Runtime exception)

checked or

RTE

B() { 5/0 }

A() { try { B() } catch } main() { A() }

Handled automatically

CTE

B() throws Exception { throw new Exception() }

A() { try { B() } catch } main() { A() }

Handled

9.3) Checked Exception | Finally Block

- compile time exception (compiler knows about this exception)
- how to handle exception
  (a). use try-catch
  or
  (b) throw the exception. (throws)

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class UserInput {
    psvm main() throws Exception {
        BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine());
    }
}
```

↑ exception is thrown from here

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
```
↳ resources must be closed. Or else it occupies memory

```
public class UserInput {
    psv main(String[] args) throws Exception {
        int n=0;
        sysout("Enter a number");
        BufferedReader br = null;
        try {
            br = new BufferedReader(new
                    InputStreamReader(System.in));
            n = Integer.parseInt(br.readLine()); //145
        }catch(Exception e){
            sysout(e);
        }finally {
            br.close();
        }
        sysout("Entered number is "+n));
    }
}
```

try catch

"throws Exception"

→ Require "throws Exception" because close() itself is throwing an exception.

If you do not want to use "throws" then use try catch for br.close()

Outputs

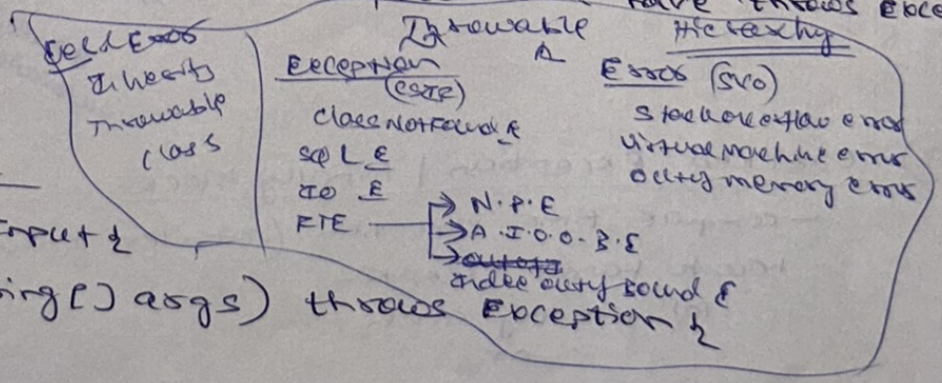
```
Enter a number
45↵
Entered number is 45
```

Outputs

```
Enter a number
abcd ↵
<< NumberFormat Exception >>
```

Note:

```
try {
}finally{
}
```
→ works fine in above code, because we have "throws Exception"



9.4) try with resource

```
public class UserInput {
    psv main(String[] args) throws Exception {
        int n=0;
        try(BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in))
        {
            n = Integer.parseInt(br.readLine());
        }
    }
}
```

will close the resource automatically

Use "catch()" or "throws Exception"

9.5) Exception Handling - user defined

(a)
```
public class ExceptionDemo {
    psv main() {
        try {
            int k = 8/9;
            if (k==0) {
                throw new Exception();
            }
        } catch (Exception e) {
            syout("Error");
        }
    }
}
```

Output:
Error

(b)
```
package com.telusko;
public class TeluskoException extends Exception {
    public TeluskoException(String s) {
        super(s); // pass the message to constructor of Throwable
    }
}
```

```
package com.telusko;
public class ExceptionDemo {
    psv main() {
        try {
            int k = 8/9;
            if (k==0) { // throw new TeluskoException();
                throw new TeluskoException("custom error");
            }
        } catch(TeluskoException e) { // or catch(Exception e)
            syout("Error : " + e.getMessage());
        }
    }
}
```

Output:
Error : custom error
// Error : null

get message from Throwable
ie "detailMessage" in Throwable class

```
public Throwable() {
    fillInStackTrace();
}

public Throwable(String message) {
    fillInStackTrace();
    detailMessage = message;
}
```

```
public class Exception extends Throwable {
    public Exception() {
        super();
    }
    public Exception(String message) {
        super(message);
    }
}
```

```
public class Throwable implements Serializable {
    private String detailMessage;
```

9.6) Users input using scanner
- no need to catch exception
- no need to parse string

```
import java.util.scanner;
Public class Userinput{
   Public static void main (String[]args){
      int n=0;
      sysout("Enter a number");
      Scanner sc = new scanner (System.in);
      n = sc.nextint(); // Fetch Integer value.
      sysout(n);
   }
}
```

Adv: - no need to ~~code~~ handle exception
- no need to convert string to Integer
  ie Integer.parseInt(str s);
- scanner also has other method. check them

Outputs
Enter a number
78↵
78

10) Multithreading in Java

- process → Thread1, Thread2 --- Threadn
- "main" → default thread
- two ways to create Threads
  (a). extend "Thread" class
  (b). implement "Runnable" interface. This is preferred as it supports multiple inheritance

```
class A
{

}

class myThread extends A, Thread
{
    int[] values = {1,2,3,4};
    public void run(){
        for(int i=0;i<5;i++){
        }    System.out.println("Hello");
    }
}
```

✗ Not supported by Java.

```
class myThread extends A implements Runnable
{
    int[] values = {1,2,3,4};
    public void run(){
        for(int i=0; i<5;i++){
        }    System.out.println("Hello");
    }
}
```

✓ Multiple inheritance

At some point, T1 & T2 reaches scheduler at the same time. Scheduler picks random thread T1 or T2 for execution. So, it may print Hello Hello or Hi Hi. To avoid, add delay at the start itself

10.1) Thread class

```
class Hi extends Thread{
    public void run(){
        for(int i=0;i<5;i++){
            System.out.println("Hi");
            try{ Thread.sleep(1000);
            }catch(Exception e){}
        }
    }
}
```

Thread1 — Thread block

```
class Hello extends Thread{
    public void run(){
        for(int i=0;i<5;i++){
            System.out.println("Hello");
            try{ Thread.sleep(1000);
            }catch(Exception e){}
        }
    }
}
```

Thread2 — Thread block

```
① . public class ThreadDemo{
    psvm(){
        Hi obj1 = new Hi();
        Hello obj2 = new Hello();
        obj1.start();
        try{ Thread.sleep(10);}
        catch(Exception e){}
        obj2.start();
    }
}
```

start() method is from Thread class

output:

```
Hi
Hello
Hi
Hello
```

(or)

```
Hi
Hello
Hi
Hello
```

(or)

```
Hi
Hello
Hi
Hello
```

10.2) Multithreading using Runnable interface

```
class Hi implements Runnable {
    public void run(){
        for(int i=0; i<5; i++) {
            System.out("Hi");
            try { Thread.sleep(1000);
            }catch (Exception e) {}
        }
    }
}

class Hello implements Runnable {
    public void run(){
        for(int i=0; i<5; i++) {
            System.out("Hello");
            try { Thread.sleep(1000);
            }catch (Exception e) {}
        }
    }
}

public class ThreadDemo {
    psv main() {
        Runnable obj1 = new Hi();
        Runnable obj2 = new Hello();
        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);
        t1.start();
        try { Thread.sleep(10);
        }catch (Exception e) {}
        t2.start();
        //System.out("Bye");
    }
}
```

outputs:

```
Hi
Hello
Hi
Hello
Hi
Hello
Hi
Hello
```

Printed by "main" Thread as it is free

```
1 [ Hi
    Hello
2 [ Hi
    Hello
3 [ Hi
    Hello
4 [ Hi
    Hello
5 [ Hi
    Hello
```

⇒ Solved using Join & isAlive refer 10.4

Note:

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

→ can use Lambda expression

P.T.O

10.3) MultiThreading Lambda Expression

```
Public class ThreadDemo {
    PSV main() {
        Thread t1 = new Thread ( () ->
        {
            for (int i=0; i<5; i++) {
                sysout ("Hi");
                try { Thread.sleep (1000); } catch (Exception e) { }
            }
        });

        Thread t2 = new Thread ( () ->
        {
            for (int i=0; i<5; i++) {
                sysout ("Hello");
                try { Thread.sleep (1000); } catch (Exception e) { }
            }
        });

        t1.start();
        try { Thread.sleep (100); } catch (Exception e) { }
        t2.start();
    }
}
```

---

class "Hi" can be removed

```
class Hi implements Runnable {
    Public void run() {
        for (int i=0; i<5; i++) {
            sysout ("Hi");
            Thread.sleep (1000);
        }
    }
}
```

→

```
Runnable obj1 = new Runnable ()
{
    Public void run() {
        for (int i=0; i<5; i++) {
            sysout ("Hi");
            Thread.sleep (1000)
        }
    }
};  // Remove only curly bracket & retain ;
```

(Anonymous class)

↓

```
Runnable obj1 = () ->
{
    for (int i=0; i<5; i++) {
        sysout ("Hi");
        Thread.sleep (1000);
    }
};
```

Replace directly.

```
Runnable obj1 = new Hi();
Thread T1 = new Thread (obj1);
```

↓

```
Thread T1 = new Thread ( () ->
{
    for (int i=0; i<5; i++) {
        sysout ("Hi");
        Thread.sleep (1000);
    }
});
```

10.4) Join | IsAlive ⇒ MultiThreading

```
public class ThreadDemo{
    psvm main(){
        Thread t1 = new Thread( ()->
        { for(int i=0; i<5; i++){
            sysout("Hi");
            Thread.sleep(1000); // use try catch here
        }
        });

        Thread t2 = new Thread( ()->
        {
            for(int i=0; i<5; i++){
                sysout("Hello");
                Thread.sleep(1000); //use try catch
            }
        });

        t1.start();
        try{ Thread.sleep(10); }catch(Exception e){ }
        t2.start();

        //sysout(t1.isAlive());

        t1.join();
        t2.join();

        //sysout(t1.isAlive());
        sysout("Bye");
    }
}
```

output:

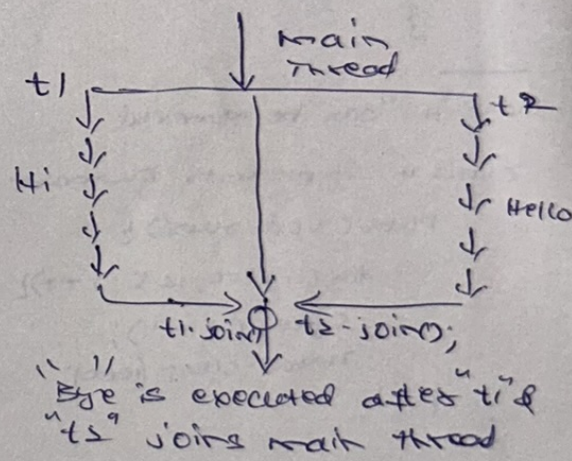
```
Hi
//false
Hello

Hi
Hello

Hi
Hello

Hi
Hello

Hi
Hello
//false
Bye
```



10.5) Thread names | Thread Priority:

- How to get Thread Name

```
sysout(t1.getName()); // default -> Thread-0
sysout(t2.getName()); // default -> Thread-1
```

- How to set Thread Name

(a) `t1.setName("Hi Thread");`
`sysout(t1.getName()); // Hi Thread`

(b). In Lambda function:

```
Thread t1 = new Thread( ()->
    { for(int i=0; i<5; i++){
        sysout("Hi");
        Thread.sleep(1000);
    }
}, "Hi Thread");
```

```
Runnable obj = new Hi();
String str = "Hi Thread";
⇒ Thread T1 = new Thread(obj, str);
```

Thread priority :

(a) Lowest priority → 1 (Thread.MIN_PRIORITY)
Highest priority → 10 (Thread.MAX_PRIORITY)
Normal (default) priority → 5 (Thread.NORM_PRIORITY).

(b). Print Thread priority
```
sysout(t1.getPriority());
```

(c). set priority
```
t1.setPriority(1); or t1.setPriority(Thread.MIN_PRIORITY);
t2.setPriority(10); or t1.setPriority(Thread.MAX_PRIORITY);
```

(c). get thread priority in thread itself
```
Thread.currentThread().getPriority();
```

```
Public class ThreadDemo {
    psv main() throws Exception {
        Thread t1 = new Thread(() ->
        {
            for(int i=0; i<5; i++){
                sysout("Hi " + Thread.currentThread().getPriority());
                Thread.sleep(1000);
            }});
        Thread t2 = new Thread(() ->
        {
            for(int i=0; i<5; i++){
                sysout("Hello " + Thread.currentThread().getPriority());
            }});
        [ t1.setPriority(Thread.MIN_PRIORITY);
          t2.setPriority(Thread.MAX_PRIORITY);
        [ sysout(t1.getPriority());
          sysout(t2.getPriority());
        [ t1.start();
          Thread.sleep(10) // try catch
          t2.start();
        [ t1.join();
          t2.join();
          sysout("Bye");
        [ sysout(t1.isAlive());
          sysout("Bye");
        }
}
```

Outputs

```
1
10
[ Hi 1
  Hello 1
[ Hi 2
  Hello 2
[ Hi 3
  Hello 3
[ Hi 4
  Hello 4
[ Hi 5
  Hello 5
[ false
  Bye
```

check output order is video of eclipse

10.6) Synchronized keyword

- Multiple threads can access the same method at the same time → NOT THREAD SAFE → Use 'synchronized' keyword for the method so that the thread waits until another thread completes execution.

```
class Counter {
    int count;
    public synchronized void increment() {
        count++;
    }
}
```

At some point, count is 600 for t1 & t2 also gets count value as 600 & count will be 601 instead of 602 - TO avoid this use Synchronized keyword so that t1 can wait until t2 execution & vice versa

OUTPUT:
1025 etc.
1430 etc but never 2000

```
public class SyncDemo {
    public static void main() throws Exception {
        Counter c = new Counter();
        Thread t1 = new Thread (new Runnable()
        {
            public void run() {
                for (int i=0; i<5; i++) {
                    c.increment();
                }
            }
        });
        Thread t2 = new Thread (new Runnable()
        {
            public void run() {
                for (int i=0; i<n; i++) {
                    c.increment();
                }
            }
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out("Count: " + ~~count~~ c.count);
    }
}
```

Output:

2000

Alternate: (If you don't want to use synchronized keyword. Rest are all same)

```
class Counter {
    AtomicInteger count = new AtomicInteger(); // AtomicBoolean
                                               // AtomicLong
                                               // AtomicIntegerArray
                                               // AtomicLongArray etc.
    public void increment() {
        count.incrementAndGet()      // count++
    }
```

11) Collections & Generics

collection interface
↑
List

- `int a[] = new int[4];` → size is always fixed
- `List<Integer> list = new ArrayList<Integer>();` // `list.add(1);`
  `list.add(1);` → can contain duplicates.

- `Set<Integer> number = new HashSet<Integer>();` → Not sorted
  `Set<Integer> number = new TreeSet<Integer>();`
  ↑ No duplicates ↓ Always sorted // `set.add(1)`

- `Map<Integer, String> m = new HashMap<Integer, String>();`
  // `map.put(5, "five");`

11.1) Iterator interface:

```
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;
Public class CollectionDemo {
    PSV main() {
        Collection values = new ArrayList();
        values.add(4);
        values.add(6);
        values.add(9);

        Iterator it = values.iterator();

        while (it.hasNext()) {
            SysOut(it.next());
        }
    }
}
```

Output:
4
6
9

// suppose → `sysout(values);` → [4, 6, 9]

Note: check 'collection' & 'Iterator' interfaces : ctrl+click.

11.2) List Interface

```
import java.util.List;
public class CollectionDemo {
    PSV main() {
        List values = new ArrayList();
        values.add(4); // Integer (V) = new Integer(4);
        values.add(6);
        values.add(9) → objects
        values.add(2, 2)
                   ↑ index   ↑ element
        values.add("x");
        Iterator it = values.iterator();
        while(it.hasNext()){
            sysout(it.next()) // 4 6 2 9 x
        }
```

```
for(int i=0; i< values.size(); i++){
    sysout(values.get(i));
}          // 4 6 2 9 x

for(Object o : values){
    sysout(o);
} // 4 6 2 9 x
```

11.3) Generics with List

```
Public class collectionDemo{
  psv main(){                                   Generics
                                                  ↑
    List<Integer> values = new ArrayList<Integer>();
    values.add(1);                                ↑
    // values.add("4");  → ERROR                  collection
  for(Object o : values){
    sysout(o);
  }
}
```

collection ↙

How to remove an entry in list or map

11.4) Collections class

This example sorts based on first digit
404, 908, 639, 265 → 265, 404, 639, 908

```
public static int compare(int x, int y){
  return (x<y) ? -1 : ((x==y) ? 0 : 1);
}
```

↓ Integer class

```
Public class collectionDemo{
  psv main(){
    List<Integer> values = new ArrayList<>();
    values.add(8);
    values.add(1);
    values.add(6);
    Collections.sort(values);   // 1, 6, 8
    Collections.reverse(values); // 8, 6, 1
    Collections.shuffle(values); // 1, 6, 8, etc.
                                 // 6, 1, 8
  }
}
```

This example sorts in default

Note:
- We can sort, reverse, shuffle etc because List is mutable ie. its value can be changed

11.5) Comparator Interface

```
Public class collectionDemo{
  psv main(){
    List<Integer> values = new ArrayList<>();
    values.add(404);
    values.add(908);
    values.add(639);
    values.add(265);
```

class myComparator implements Comparator{
public int compare(i, j){ return ... }}

```
Comparator<Integer> c = new myComparator();
```

// sort based on last digit - custom implementation
⇒ 404, 908, 639, 265 → 404, 265, 908, 639

Functional Interface

Anonymous class:

```
Comparator<Integer> c = new Comparator<Integer>(){
  public int compare(Integer i, Integer j){
    return i%10 > j%10 ? 1 : -1;
  }
};
Collections.sort(values, c);
```

or

Lambda function:

```
Comparator<Integer> c = (i, j) -> i%10 > j%10 ? 1 : -1;
Collections.sort(values, c)
```

or

```
Collections.sort(values, (i,j) -> i%10 > j%10 ? 1 : -1);
}
```

11.6) Comparator Interface

```
class Student implements Comparable<Student> {
    int rollNO, marks;
    String name;

    public Student(int rollNO, String name, int marks) {
        super();
        this.rollNO = rollNO;
        this.name = name;
        this.marks = marks;
    }

    @Override
    public String toString() {
        return "Student [rollNO=" + rollNO + ", marks=" + marks +
                ", name=" + name + "]";
    }

    public int compareTo(Student s) {
        return name.length() > s.name.length() ? 1 : -1;   → sort by name
    }
}
```

(implementing method for 'compareTo' from comparable interface)

" marks " " → sort by marks

" rollNO " " → sort by rollNO

```
public class CollectionDemo {
    PSVM main() {
        List<Student> student = new ArrayList<>();
        student.add(new Student(2, "mahesh", 55));
        !

        Collections.sort(student);
        for(Student s : student) {
            SysOut(s);
        }
    }
}
```

NOTE:

- Now, if you want to sort by marks although sort implementation is for name, use below technique

```
Collections.sort(student, (i, j) -> i.marks > j.marks ? 1 : -1);
```

- Comparator is overriding comparable object.
- whenever a class's object is to be compared, that class must implement Comparable interface

11.7) Set Interface

```
import java.util.Set;
import java.util.TreeSet;
import java.util.HashSet;

Public class setDemo {
    Pubic static void main(){
        Set<Integer> values = new HashSet<Integer>();
        values.add(6);
        values.add(2);
        values.add(5);
        values.add(6); → value is not added as its duplicate
        for(int i : values){
            sysout(i);
        }
    }
}
```

Outputs:
6
2
5

Notes:
Set<Integer> values = new TreeSet<Integer>(); → sorts
Output:
2
5
6

11.8) Map Interface

```
import java.util.Hashtable; // Hashtable is synchronized → Thread safe
import java.util.HashMap; // → Not thread safe
import java.util.Map;

Public class MapDemo{
    PSV main(){
        Map<String, String> map = new HashMap<>();   ↗ <String, String>
        map.put("one", "Bangalore");
        map.put("two", "London");
        map.put("three", "Lisbon");
        Set<String> keys = map.keySet();
        for(String key : keys){
            sysout(key + " " + map.get(key));
        }
    }
}
```

Outputs
one Bangalore
two London
three Lisbon

Notes
(String, String)
Map<String, String> map = new Hashtable<>(); → Thread safe

TOPICS

App 1

- volatile
- JVM Architecture, memory models, GC
- locks in threads
- SQL queries (top 2nd, etc)
- Exceptions
- Java (8/11) New Features
- Design patterns

HashMap:
one null key, multiple null values allowed

HashTable:
No duplicate keys allowed

spring.jpa.hibernate.ddl-auto=
create
validate
update
create-drop

webhook
payload size (postman etc.

How to secure endpoints

palindrome
reverse string.

IOC
D-I
BeanFactory

① List.of() ?

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> res = numbers.parallelStream()
        .filter(p -> p > 2)
        .map(p -> p * p)
        .sorted()
        .collect(Collectors.toList());
```

```
String[] str = new String[]{"a", "b", "c", "d"};
```

② Optional<String> optional = optional.ofNullable(str[2]);

```
if (optional.isPresent()) {
    System.out.println(optional.get());
}
```

optional.of() → doesn't handle NPE

→ Java: final(finally)finalize()
- Why we need F.I
- Comparator & Comparable
- F.I
- try catch finally features in try block

→ Remove duplicates in linked list

→ Spring Boot: Actuators, Userfiles, Aspects,
How to initialize db, query startup
JPA repository,
DB exceptions(rare)
How do you handle exceptions.

→ Docker: Exec command.

→ Junit: How to mock static method,
How to test void method.

→ How to store passwords

→ OAuth2.

Design patterns: Any two

Add two numbers

```
import java.util.Stack;

Public static Node addTwoNumbers (Node list1, Node list2){
(a) ⇒ Stack <Integer> stack1 = new Stack<>();
      Stack <Integer> stack2 = new Stack<>();
      while (l1 != null) {
          stack1.push(l1.data);
          l1 = l1.next;
      }
      while (l2 != null) {
          stack2.push (l2.data);
          l2 = l2.next;
      }

(b) ⇒ int carry = 0
      Node result = null
      while(!stack1.isEmpty() || !stack2.isEmpty()) {
      int a=0, b=0;
      if (!stack1.isEmpty()){          if (!stack2.isEmpty()){
          a = stack1.POP();                b=stack2.POP();
      }                                }
      int total = a + b + carry;
      Node temp = new Node (total % 10);
      carry = total/10;
      if (result == null) {
          result = temp;
      }else {
          temp.next = result;
          result = temp;
      }
      }

(c) ⇒ if (carry != 0) {
          Node temp = new Node (carry);
          temp.next = result;
          result = temp;
      }

(d) ⇒ return result;
    }
```

Questions

① · continue, break in nested loops

② · can i use constructor to initialize static variable
[static block
constructor block]

③ · public or private for static variables?
which one to use?
basics
∴ Employee.CEO = "LARRY";

④ · Lambda function → parameters.
() → sysout. How to pass params in this case?
(12|12b)

⑤ · can we have final method in interface (13)

---

⑥ · Java 8 Features

⑦ · Different type of Functional interface

- Diff. btw arraylist & linkedlist
- scopes in spring boot
- How to read property value
- streams
- hashcode() method purpose in case of overriding.

→ Spring security
→ Actuators / AOP

Lambda Function Notes

(A)

(1). @FunctionalInterface

interface HotelFilter {

boolean test (Hotel hotel);

(B) }

(2) class HotelLessThanThousand implements HotelFilter {

@Override

boolean test (Hotel hotel) {

return hotel.price < 1000;

}

}

(3) class Hotel:LessThan2000

(1)(2)

(C)

class HotelWrapper {

static List<Hotel> filter (List<Hotel> hotel,

HotelFilter hotelFilter) {

List<Hotel> res = new ArrayList<>();

for (Hotel hotel : hotels) {

if (hotelFilter.test (hotel)) {

res.add (hotel);

}}

return hotels;

}}

(D)

main class

```
List<Hotel> hotelLessThan1000 =
        HotelFilter.filter(hotels,
            new HotelsLessThanThousand());
```

or

```
List<Hotel> hotelLessThan1000 =
    HotelFilter.filter(hotels,
        new HotelFilter() {
            @Override
            public boolean test(Hotel hotel) {
                return hotel.price<1000;
            }});
```

Anonymous class eliminates step (2)

or

```
List<Hotel> hotelLessThan1000=
    HotelFilter.filter(hotels, hotel-> hotel.price<1000);
```

↑ functor.

or

~~List<Hotel> hotelLessThan:~~

If you want to remove functional interface ie (1) HotelFilter, use Predicate. No need to create interface.

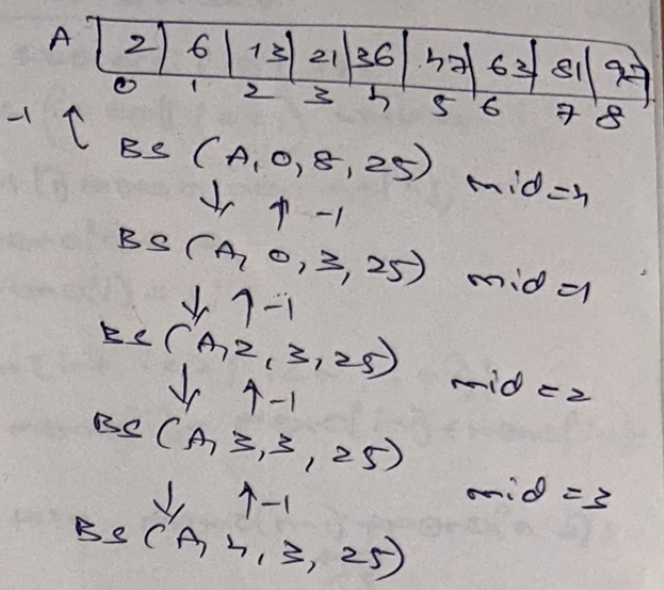
Eg) `List<Hotel> filter(List<Hotel> hotels, Predicate<Hotel> predicate);`

`predicate.test(hotel)` no other changes

Binary search

```
BinarySearch (A, low, high, x) {
  if (low > high) return -1
  mid = low + (high - low)/2
  if x == A[mid]
      return mid
  else if (x < A[mid])
      return BS(A, low, mid-1, x)
  else
      return BS(A, mid+1, high, x)
}
```

$O(\log n)$



Iterative

T.C: $O(\log n)$

```
public int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
      int mid = (low + high) / 2
      if (key > a[mid]) {
          low = mid + 1;
      } else if (key < a[mid])
          high = mid - 1;
      } else {
          return mid
      }
    return -1;
}
```

Linear search: $O(n)$

Binary search: $O(\log n)$

Bubble sort

a b c d (length = 4)
0 1 2 3 3 iterations

(a) (i = 0; i < 3; i++)
(j = 0; j < 3; j++)

(b). (i = 1; i < 3; i++)
(j = 0; j < 2; j++)

(c). (i = 2; i < 3; i++)
(j = 0; j < 1; j++)

(i = 0; i < n-1; i++){
(j = 0; j < n-i-1; j++){
if arr[j] > arr[j+1]{
swap
}

T.C
selection sort O(n²)
~~Bubble~~ sort O(n²)
~~Insertion~~ sort O(n²)
heap sort O(n · log n)
quick sort O(n²)
merge sort O(n · log n)
radix sort O(n k)

Insertion sort

a [b] c d Length = 4
0 1 2 3 3 iteration

for (i = 1; i < n; i++){
(a). key = arr[i]; } save value & prev. position
j = i - 1;
(b). while (j >= 0 && arr[j] > key)
arr[j+1] = arr[j];
j = j - 1;
}
(c). arr[j+1] = key;
}

sorted Array | unsorted Array

(a). 9 [8] 5 2
0 1 2 3
(b). 8 9 [5] 2
8 9 5 2
(b). 8 9 [5] 2
5 8 9 2
(c). 5 8 9 [2]
2 5 8 9

repeat

```
int arr[] = {64, 34, 25, 12}
void bubbleSort(int arr[]){
    int n = arr.length;
    for(int i=0; i<n-1; i++){
        for int j=0; j<n-i-1; j++)
        if (arr[j] > arr[j+1])
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
    }
}
```

O(n²)

PTO

```
void insertionSort(int arr[]){
    int n = arr.length;
    for(int i=1; i<n; i++){
        int key = arr[i];
        int j = i-1;
        while (j>=0 && arr[j] > key)
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

O(n²)

compare 64 & 25 → 25 → 1
25 & 12 → 12 →
1

Selection sort

S.A | U.S. Array

0	1	2	3	4	→	SA	USA			
64	25	12	22	11	→	11	25	12	22	64

64 & 11

11 | 25 12 22 64 → 11 | 12 25 22 64

25 & 12

11 12 | 25 22 64 → 11 12 22 | 25 64

22 & 25

11 12 22 | 25 64

11 12 22 25 | 64

(a) (64) & 25 ⇒ min = 25 ; pos = 1
(25) & 12 ⇒ min = 12 ; pos = 2
(12) & 22 ⇒ — " —
(12) & 11 ⇒ min = 11 pos = 4
swap 64 & 11

SA 11 | USA 25 12 22 65

```
for (i=0; i<n-1; i++) {
  (a). min-idx = i;
    for (j=i+1; j<n; j++) {
        if (arr[j] < arr[min-idx]) }
        }  min-idx = j;
    }
  (b). swap
    int temp = arr[min-idx];
    arr[min-idx] = arr[i];
    arr[i] = temp
}
```

Find min element in unsorted Array

```
void selection sort (int[] arr) {
    int n = arr.length;
    for (int i=0; i<n-1; i++) {
      (a). int min-idx = i;
        for (int j = i+1; j<n; j++) {
            if arr[j] < arr[min-idx] {
                min-idx = j;
            }
        }
      (b). int temp = arr[min-idx];
        arr[min-idx] = arr[i];
        arr[i] = temp;
    }
}
```

$O(n^2)$

Bubble sort

⇒ 6 8 1 | 2
6 8 1 | 2
6 1 8 | 2
6 1 2 | 8

⇒ 6 1 | 2 8
1 6 | 2 8
1 2 6 8

⇒ 1 | 2 6 8
1 2 6 8

```
int isPrime(int x){
    if (x==0||x==1) return false;
    for(int i=2; i<=√x; i++){
        if (x%i==0)
            return false
    }
    return true
```

```
}
return (n==0||n==1)? 1 : n*fact(n-1)
int fact(int n){
    if (n<0) return -1
    else if (n==0) return 1;
    else return n*fact(n-1);
}
0! = 1 ; 1! = 1
n! = n * (n-1)!
```

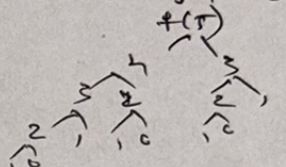
```
int fib(int n){
    if (n==0||n==1) return n;
    return fib(n-1)+fib(n-2)
}
```

TD1

```
int fibonacci(int n){
    int[] memo = new int[n+1];
    return fib(n, memo)
}
int fib(int n, int[] memo){
    if (i==0||i==1) return i;
    if (memo[i]==0)
        memo[i] = fib(n-1, memo)+
                  fib(n-2, memo)
    return memo[i];
}
```

Bottom-up approach

```
int fibonacci(int n){
    if (i==0||i==1) return i;
    int[] memo = new int[n];
    memo[0] = 0;
    memo[1] = 1;
    for(int i=2; i<n; i++){
        memo[i] = memo[i-1]+memo[i-2];
    }
    return memo(n-1)+memo(n-2);
```




```
int min_index=0;
for(int i=1; i<n; i++)
    if (a[i] < a[min_index])
        min_index = i;
}
int max_index=0;
for(int i=1; i<n; i++){
    if (a[i] > a[max_index])
        max_index=i;
}
swap(a[min_index], a[max_index]);
```