Java Content -TELUSKO

==========

- Variables and Data Types
- Identifiers
- Type Casting
- Operator
- Conditional Statements
- Iterative Statements
- Switch Statement
- Number system
- Naming Convention
- Class and Object
- JVM Memory
- Methods
- this keyword
- Scanner Class
- OO Programming Features
    - Polymorphism [Overloading]
    - Encapsulation
- Constructor
- Static keyword
- Inheritance
- Upcasting and Down casting
- Polymorphism [Overriding]
- Access Modifiers
- Abstraction
- Final keyword
- Java Program Execution
- Static keyword
- Interface
- Anonymous Inner Class
- Arrays
- String Class
- Interface and Lambda Expression
- Exception handling
- Packages
- Collection
- Generics
- Map
- ForEach
- Stream API
- Java 8 and + features
- IO
- Multithreading

Variables and Data Types

==========

->To Store data, we use variables and to specify what type of data we are storing in the variables then we need to use Data types.
->It is used for memory allocation.

Types of Variables

Instance variable

->Declared outside method, block, constructor and inside the class.
->Has default value
->Can be accessed using reference variable

Local variable

->Declared inside method, block, constructor, loop.
->Not have default value, should be initialized before use.

Static variable

->Declared outside method, block, constructor and inside the class with keyword static.
->Has default value
->Can be accessed using reference variable and class name

Primitive Data type

->-2^(n-1) to 2^(n-1) -1
->If exceeds the range then Compiler will throw an error.
->Default data type for whole number, Java consider is int, If we are performing any operation on 2 or more numbers then the result data is by default is int data type.
->Default data type for float number, Java consider is double, If we are performing any operation on 2 or more numbers then the result data is by default is double data type.
->Bases on UTF, Unicode representation above memory representation.

byte

1 byte of memory is allocated
Default Value: 0
Ex:
byte b=22;

short

2 byte of memory is allocated
Default Value: 0
Ex:
short s=33;

char
2 bytes of memory is allocated
Ex:
char c='A'; //A
char c=65; //A

int
4 bytes of memory is allocated
Default Value: 0
Ex:
int i=345;

long
8 bytes of memory is allocated
Default Value: 0
Ex:
long l=3456l;
long l=12345L;

float
4 bytes of memory is allocated
Default Value: 0.0
7 digit precession
Ex:
float f=12.3f;
float f=23.44F;

double
8 bytes of memory is allocated
Default Value: 0.0
14 digit precession
Ex:
double f=12.3;
double f=23.44;

Boolean
1 bit of memory is allocated
Default Value: false
Ex:
Boolean b=true;

Identifiers

==========

->Start with letters, _ and $

->Can contains letters, digits. _ and $

->No use of Keyword or Reserve words

->Give meaningful names

Type Casting

==========

->Converting data of one type to another type.

->Types

Implicit: Lower to Higher implicitly by JVM

Ex:

```
byte b=10;
int i=b;
```

Explicit: Higher to Lower explicitly by developer using cast operator

Ex:

```
int i=10;
byte b=(byte) i;
```

->Type checking: Compiler

Type Casting: JVM

Type casting chart

byte->short->int->long->float->double

^

char

Operator

==========

->Increment/Decrement operator

int a=5;

a=a+1;

OR

a=a++;

->a++: assign and increment

++a: increment and assign

Types of Operators

Arithmetic Operator: + - * / %

Ex:

-20%7=-6

10%30=10

10%-3=1

Relational Operator:> < >= <= == !=

results Boolean value

Logical Operator: && || !

Combines two or more relational operators

Shortcut Operator: & |

Conditional Statements

==========

->if/if-else/if else-if else/ternary operator

->Condition can be simple relational operator or combination of Logical and relational operator

->Nested conditional are supported

```
if(condition)
{....}

if(condition)
{.....}
else
{.....}

if(condition)
{.....}
else if(condition)
{.....}
else
{......}
```

->Ternary operator

Syntax:

var = condition ? exp1 : exp2;

condition true, var=exp1 else var=exp2

Ex:

```
int a=(10>20)?10:20;
sop(a+" is big");
```

->exp can be another ternary operator

Iterative Statements

==========

->for loop/while loop/do while loop

for(initialization; condition; increment or decrement)
{.....}

initialization
while(condition)
{
.....
increment or decrement
}

initialization
do
{
.....
increment or decrement
}while(condition);

Switch Statement

==========

Syntax:

```
switch(exp)
{
case exp1:
statement;
break;

case exp1:
statement;
break;

default:
statement;

}
```

->exp can contains: byte, short, int, String

->exp1,2,3,.. should be constant or literals

->If we not add break then all below cases after satisfying condition will be executed.

Ex:

```
public class SwitchDemo {
   public static void main(String[] args) {
      int a = 40;

      switch (a) {
         case 10:
            System.out.println("Appu");
            break;

         case 20:
            System.out.println("Raghu");
            break;

         case 30:
            System.out.println("Shivu");
            break;

         default:
            System.out.println("Sharukh");
      }
   }
```

Number system

==========

->Always output will be in decimal number format

->**Octal**

Prefix: O or o

Allowed: 0,1,...7

Ex:

int a=O45;

sop(a); //37

->**Hexadecimal**

Prefix: OX or ox

Allowed: 0,1,...9,A,B,C,D,E

Ex:

int a=OX45;

sop(a); //69

->**Binary**

Prefix: OB or ob

Allowed: 0,1

Ex:

int a=OB0100101;

sop(a); //37

Naming Convention

==========

->Class, Interface, Enum names should start with uppercase and subsequent letter should starts with uppercase

->Package name should be lowercase

->Variables and Method names should be in Camel case.[start with small and subsequent letter should be in uppercase]

->Constant should be all in uppercase.

->No space allowed, can use _

Class and Object

==========

->Class is a Blueprint

->Object is the instance of the Class

->reference variable contains reference value of the object and points to the created object.

->Creation of object is knows as Instantiation

Class creation:

class className

{

constructors();

instance/static variables;

instance methods();

}

Object Creation:

className refVarName=new className();

->Exe flow

.java->COMPILER-->.class file-->JVM--->Output

Compile: Compiler during Compilation time

Execute: JVM during Runtime inside JRE

JVM Memory

==========

->JVM exe inside JRE environment

->Method area/Stack area/Heap area/PC Register/Nature stack

new className();

->New memory will be created inside Heap area based on instance variable.

->Assign new value known as reference value for the object.

->Store Instance variable data, if not default value will be assigned.

Stack Area:

->When any method start to execute, stack trace for it will be created in stack area. Once it got executed, stack trace will be removed.

->Reference and Local variable will be stored in Stack area.

->Once stack trace is removed, local and reference variable will be removed.

Heap Area:

->Memory for the Object will be allocated from heap area and it contains Instance and Static variable data.

->When object do not have any reference variable then GC will remove it from the heap memory.

Methods

==========

->Can be accessed by reference variable, class name or directly.

[accessModifiers] returnType methodName([parameters])
{
.....
[return value];
}

Note: When same method call itself finally we get StackOverflowError exception

this keyword

==========

->Used to refer current class instance member

Scanner Class

==========

```
Scanner scanner=new Scanner(System.in);
String name=scanner.next();
```

Object Oriented Programming Features

==========

Class/Object/Encapsulation/Data hiding/Abstraction/Inheritance/Polymorphism

Polymorphism [Overloading]

->Method overloading and Method overriding

Method overloading

->2 or more methods in the class having same name and different parameter list[type or count].

->Which method to be called is decided at compile time itself. It is also knows as Compile time polymorphism.

Ex:

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

int add(int a, int b, int c) {
   return a + b + c;
}
```

->It is not possible to create 2 method with same name and parameter list even if return type is different.

Ex: Not allowed

```
float add(int a, int b)
{
   return a+b;
}

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

Encapsulation

->Make variable in the class as private and method as public.
->Binding variables and method together as single unit.
->Access variables through methods for increasing security.
->getter() used to get the value and setter() is used to set the value for the instance variables.

Ex:

```
class Demo1 {
   private int age;
   private String name;

   int displayAge() {
      return age;
   }
   String displayName() {
      return name;
   }
   void setAge(int age) {
      this.age = age;
   }
   void setName(String name) {
      this.name = name;
   }
}
```

Constructor

==========

->Special method having name same as Class name, No return type, Invoke when object is created, Used to initialize the instance var data.

->Can have Access modifiers.

->On each Object creation Constructor is invoked.

->Constructor can be overloaded but cannot able to overrided.

->Works similar as setter(), can also do many task similar to methods.

->If dev has not added any constructor then by default , Compiler will add default 0 argument constructor.

->Types: Default[0 Argument] / User defined [0 argument and parameterized constructor]

->Can add default, public, protected and private access modifiers

Ex:

```
class Demo2
{
   int age;
   String name;
   Demo2(String name,int age)
   {
      if(age>18)
      {
         this.age=age;
      }
      else {
         System.out.println("Invalid Age");
      }
      this.name=name;

   }
}
public class ConstructorDemo1 {
   public static void main(String[] args) {
      Demo2 d1=new Demo2("Appu",23);
      System.out.println(d1.age+" "+d1.name);
      Demo2 d2=new Demo2("Sivu",15);
      System.out.println(d2.age+" "+d2.name);
   }
}
```

OUTPUT:

```
23 Appu
Invalid Age
0 Sivu
```

->First statement of the constructor is super()
which will invoke parent class 0 argument constructor
If we add this([parameter list]); manually then super() will not be there.
->First statement can be either this or super not both.
->Always parent class should have user defined or compiler defined 0 argument constructor, if all child class constructor don't have this() or super(param list)

Ex 1: Valid since parent have 0 argument constructor

```
class A
{
   A()
   {
      System.out.println("A 0 Arg Const");
   }

   A(int a)
   {
      System.out.println("A 1 Arg Const");
   }
}

class B extends A
{
   B()
   {
      System.out.println("B 0 Arg Const");
   }

   B(int a)
   {
      System.out.println("B 1 Arg Const");
   }
}
public class ConstructorSuper {
   public static void main(String[] args) {
      B b=new B(10);
   }
}
```

OUTPUT:
A 0 Arg Const
B 1 Arg Const
Flow: child 1 argument constructor>super() of child 1 argument constructor >parent 0 argument constructor>child 1 argument constructor

Ex 2: In Valid, parent do not have 0 argument constructor

```
class A
{
   A(int a)
   {
      System.out.println("A 1 Arg Const");
   }
}

class B extends A
{
   B()
   {
      System.out.println("B 0 Arg Const");
   }

   B(int a)
   {
      System.out.println("B 1 Arg Const");
   }
}
public class ConstructorSuper {
   public static void main(String[] args) {
      B b=new B(10);
   }
}
```

OUTPUT:

ERROR

Flow: child 1 argument constructor>super() of child 1 argument constructor >parent 0 argument constructor but its not available

->If any child class have constructor then parent must have 0 argument constructor else we need to add super(paramlist) in all child class constructor to avoid parent 0 argument constructor call from child class

Ex 3: Valid, Since we added manually super(param list) in all child class constructor

```
package basicstelusko;

class A {
   A() {
      System.out.println("A 0 Arg Const");
   }
```

```
    A(int a) {
        System.out.println("A 1 Arg Const");
    }
}

class B extends A {
    B() {
        super(10);
        System.out.println("B 0 Arg Const");
    }

    B(int a) {
        super(10);
        System.out.println("B 1 Arg Const");
    }
}

public class ConstructorSuper {
    public static void main(String[] args) {
        B b = new B(10);
    }
}
```

OUTPUT:
A 1 Arg Const
B 1 Arg Const

Constructor chaining

One Constructor calls other Constructor of same class.
Case 1:
>Within the class using this keyword
should be in first line inside the constructor.
this([parameter list])

>To call parent class constructor from child class constructor
>should be in first line inside the constructor.
super([parameter list])

Ex:

```
package basicstelusko;

class Demo3 {
```

```
   Demo3()
   {
      System.out.println("Demo3 Zero Arg Const");
   }

   Demo3(int a,int b,int c)
   {
      System.out.println("Demo3 three Arg Const");
   }
}
class Demo4 extends Demo3
{

   Demo4(int a)
   {
      System.out.println("Demo4 One Arg Const");
   }

   Demo4()
   {
      //this(10);
      super(1,2,3);
      System.out.println("Demo4 Zero Arg Const");
   }

   Demo4(int a,int b,int c)
   {
      System.out.println("Demo4 three Arg Const");
   }
}
public class ConstructorChaining {
   public static void main(String[] args) {
      Demo4 d=new Demo4();

   }
}
```

OUTPUT:
Demo3 three Arg Const
Demo4 Zero Arg Const

->Execution flow
Parent class constructor
Instance var/block
Child class constructor

Main method

==========

->We can overload main method but JVM consider the standard syntax main method for execution.

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

static keyword

==========

->Can be used with variables, block or to method

->Exe flow without parent class

static block or static variable

instance block or instance variable

constructor

instance block or instance variable

constructor

static block

->

static

{

....

}

Inheritance

==========

->IS-A relation

->Re using exiting members[methods and variables] of the class instead of creating again.

->Parent Child relationship

->

parent/super/base

child/sub/derived

child class extends parent class

{

}

->Private property[var and method], constructor not participate in inheritance.

Types of methods in Inheritance

Inherited methods: Method available in parent class and used without any modification in child class.

Overridden methods: Parent class method with new body in child class.

Specialized methods: Child class methods with its own body

Accessing methods in Inheritance

Parent obj=new Child(); //Upcasting

obj-> can access inherited methods[parent class methods], overrided method[but we get body of child class].

Cannot able to access specialized methods, if need to access need to do type casting[Down casting].

Parent obj=new Parent();

obj->can access inherited methods.

Cannot able to access specialized methods, if need to access need to do type casting.

Child obj=new Child();

obj->can access inherited method, overrided method and specialized methods.

Ex:

Ex:

```
class X {
   void add() {
      System.out.println("parent add");
   }

   void sub() {
      System.out.println("parent sub");
```

```
    }
}

class Y extends X {
    void add() {
        System.out.println("child add");
    }

    void mul() {
        System.out.println("child mul");
    }
}

public class UpcastingDowncasting {
    public static void main(String[] args) {
        System.out.println("****NORMAL*****");
        X x = new X();
        x.add(); //Inherited method
        x.sub(); //Inherited method

        System.out.println("****NORMAL*****");
        Y y = new Y();
        y.add(); //Overridden method
        y.mul(); //Specialized method
        y.sub(); //Inherited method

        System.out.println("****UPCASTING*****");
        X xy = new Y();
        xy.add(); //Overridden method
        xy.sub(); //Inherited method


        System.out.println("****DOWNCASTING*****");
        //Y yx=new X(); //ERROR
        Y yx = (Y) new X();
        yx.add(); //Overridden method
        yx.sub(); //Inherited method
        yx.mul(); //Specialized method
        System.out.println("*********");

    }
}
```

OUTPUT:
****NORMAL*****
parent add
parent sub
****NORMAL*****
child add
child mul
parent sub
****UPCASTING*****
child add
parent sub
****DOWNCASTING*****
Exception in thread "main" java.lang.ClassCastException: class oopstelusko.X cannot be cast to class oopstelusko.Y (oopstelusko.X and oopstelusko.Y are in unnamed module of loader 'app')
	at oopstelusko.UpcastingDowncasting.main(UpcastingDowncasting.java:44)

Inheritance Types

Single/Multi level/Multiple/Hierarchal
C extends A,B ->Not allowed due to Ambiguity issue

Constructor flow

---

object creation for child class>child class cons>super()>parent class 0 argument constructor>child class constructor

Always parent class 0 argument constructor will be invoked

this() vs super()

To call constructor of same class we use this() and to call constructor of super class we use super()
Both should be inside first line of constructor only

Variable/method preference

---

local, class, super class

this vs super

---

this.varname: calls instance variable
super.varname: calls super class variable

When not to use Inheritance[HAS-A vs IS-A]

IS-A: Inheritance, if there exist is-a relation then use extends.

HAS-A: Composition, If there not exist is-a relation then else use has-a relation
Create Object of that class and call those methods.

Ex 1: HAS-A relation

```
class Phone
{
call(){...};
}

class Human
{
}
```

Human needs call() but not exist is-a relation so create Object and use

Solution:

```
class Human
{
Phone obj=new Phone();
obj.call();
}
```

Ex 2: HAS-A relation

```
class Bathtub
{
bath(){...};
```

```
}

class Home
{
Bathtub obj=new Bathtub();
obj.bath(); //Since there not exist is-a relation we use has-a relation to access method
}
```

Ex 3: iPhone 14 extends iPhone 13
It has is-a relation so we can use extends

```
class iPhone13
{
playSong(){...};
}

class iPhone14 extends iPhone13
{
playSong(); //There exist is-a relation so use extends
}
```

Upcasting and Down casting

==========

Parent p=new Child(); //Upcasting and its allowed , used to achieve Polymorphism

Child c=(Child)new Parent(); //Down casting and it throws ClassCastException exception

Ex:

```
class X {
   void add() {
      System.out.println("parent add");
   }

   void sub() {
      System.out.println("parent sub");
   }
}

class Y extends X {
   void add() {
      System.out.println("child add");
   }

   void mul() {
      System.out.println("child mul");
   }
}

public class UpcastingDowncasting {
   public static void main(String[] args) {
      System.out.println("****NORMAL*****");
      X x = new X();
      x.add(); //Inherited method
      x.sub(); //Inherited method

      System.out.println("****NORMAL*****");
      Y y = new Y();
      y.add(); //Overridden method
      y.mul(); //Specialized method
      y.sub(); //Inherited method

      System.out.println("****UPCASTING*****");
      X xy = new Y();
      xy.add(); //Overridden method
      xy.sub(); //Inherited method
```

```
        System.out.println("****DOWNCASTING*****");
        //Y yx=new X(); //ERROR
        Y yx = (Y) new X();
        yx.add(); //Overridden method
        yx.sub(); //Inherited method
        yx.mul(); //Specialized method
        System.out.println("*********");

    }
}
```

OUTPUT:

```
****NORMAL*****
parent add
parent sub
****NORMAL*****
child add
child mul
parent sub
****UPCASTING*****
child add
parent sub
****DOWNCASTING*****
Exception in thread "main" java.lang.ClassCastException: class oopstelusko.X cannot be cast to class oopstelusko.Y (oopstelusko.X and oopstelusko.Y are in unnamed module of loader 'app')
        at oopstelusko.UpcastingDowncasting.main(UpcastingDowncasting.java:44)
```

Polymorphism [Overriding]

->Same thing with different behaviour
->We can create super class reference and sub class Object
Parent p=new Child();
->Method of sub class overrides method of super class
Giving new functionality for the method of super class in child class
Runtime Polymorphism, In compile time compiler not aware of which method will be called either super or sub class method.
Create object for child class and reference for parent class, call the method.

Ex:

```
class Phone
{
   void call()
   {
      System.out.println("Calling");
   }
}

class Iphone extends  Phone
{
   void call()
   {
      System.out.println("Calling from Iphone");
   }
}
public class MethodOverridingDemo {

   public static void main(String[] args) {
      Phone obj=new Iphone();
      obj.call();
   }

}
```

OUTPUT:
Calling from iPhone

->
Allowed:
superclass reference =child class object();

Not Allowed:
sub class reference = super class object();

Rules for Method overriding

->Cannot reduce Access modifiers power
->Method prototype[heading] should be same
->@Override annotation should be used
->private method cannot be overrided
->we cannot change return type for primitive data type, but if it has non primitive data type we can change but there should be parent child relationship between the return type of parent class return type and child class return type.

Co variant return type
parent class return type display()
{....}
child class return type display()
{...}

->Cannot change argument list
->Static method cannot not be override-method hiding[we get parent method body only not child]

Ex:

```
class Phone1
{
   static void call()
   {
      System.out.println("Calling");
   }
}

class Iphone1 extends  Phone1
{
   static void call()
   {
      System.out.println("Calling from Iphone");
   }
}

public class MethodHiding {

   public static void main(String[] args) {
      Phone1 obj=new Iphone1();
      obj.call();
   }
}
```

OUTPUT:
Calling

Access Modifiers

==========

->default/public/protected/private

->Can be applied to variable, constructor and methods

public: anywhere inside the project

Protected: within package and outside package with is-a relation

default: within package

private: within class

public>protected>default>private

Abstraction

==========

->Hiding implementation from user.

->Just declare methods without body then its an abstraction.

->Can be achieved using abstract keyword and interface.

->If the body of the method is not used by any of the child class then we can make those method as abstract methods. In child class we can provide required implementation.

Abstract Method

->Method with only signature and no body/ Declared using keyword abstract/Present inside abstract class and Interface.

->Common implementation for all the child class make it as concrete method, different implementation for different child class then make it as abstract method.

Abstract class

->Class with 0 to N abstract methods and/or 0 to N concrete methods.

->Any class contains abstract methods then that class should be abstract class.

->There should be a class providing implementation for it using extends keyword.

->Cannot create object for abstract class

->Abstract class can contain constructor, constructor cannot be abstract.

->Incomplete class

->Class that extends abstract class should provide body for all the abstract methods then we can create object for the implementation class else that class should also be abstract class.

Final Keyword

==========

->Final keyword can be applied to variable/method/class

->Final class cannot eb inherited

->Final method cannot be overrided, do get inherited in child class.

->Final variable cannot be re initialized, need to be initialized, acts like constants.

->Abstract method and abstract class cannot be final

Static Keyword

==========

->Static context: static variables/static block/static methods

->Instance context: instance variables/instance block/instance methods

->Static members are not object dependant but instance members are object dependant.

Flow of execution

---

static block or variable[class loading once per execution]

instance block or variable[object creation/for each object creation]

constructor[object creation/for each object creation]

instance block or variable[object creation/for each object creation]

constructor[object creation/for each object creation]

Static variable

->Static variables can be accessed inside sm, sb,im,ib.

But instance variables can be accesses only inside ib and im.

->If any data has to be shared among all the object of the class we need to make those variable as static variables.

->Memory allocated once during class loading in Heap area

->can be accessed using class name or object reference variable

Static method

->Can be accessed using Class name or reference variable.

Static Block

->Executed once while loading .class file to memory.

Java Program Execution

==========

->Compiler compiles and generates .class file for each class, Enum, interface, abstract class.

->JVM executes Java program in JRE.

->JVM make use of Class loaders/JVM Data Areas/Execution engine

Class loaders

JVM loads the .class file to method area using Class loaders, allocates memory for static variables in heap memory and also provide default values.[executes static context during class loading]

JVM Data Areas

Method area: complete .class file will be loaded here.

Stack area: Reference variable, Local variable, Stack frame for each method execution.

Heap area: static variables, instance variables

PC register: Address of each instruction.

Native stack: 3rd party

Execution engine

Executes the program using Interpreter, JIT compiler and GC.

Interface

==========

->All methods are public abstract

->Implementation class should provide implementation for all the abstract method of the interface else that class should also be abstract class

->All variable are public static final

->Cannot create object for interface, can able to create reference variable

->One implementation class can implements any number of interface

->One interface can extends other interface

->Interface not contains Constructor

->One class can implements any number of interfaces.

->One class can extends one class and implements n number of interfaces.

->An interface can have any number of implementations.

Ex:

```
interface Car {
   String name = "Car";
   void start();
   void stop();
}

class Maruti implements Car {
   @Override
   public void start() {
      System.out.println("Starting Car");
   }
   @Override
   public void stop() {
      System.out.println("Stopping Car");
   }
}

public class InterfaceImplementationbasics1 {
   public static void main(String[] args) {
      Car car=new Maruti();
      car.start();
      car.stop();
   }
}
```

OUTPUT:

```
Starting Car
Stopping Car
```

Marker Interface

->Interface with 0 methods.

Ex:

interface I

{...}

Default method

->From Java 8, Interface can have method with body.

->We can override Default method

Ex: default void disp()

{.....}

Static method in interface

->Interface can have static method with body.

->Static method cannot be overrided, if we override then it will be consider as specialized method not overridden method.

->Static method can be called using Interface name or implementing class name.

->Static method should have body else it throws error.

Ex:

```
interface Car {
   default void display()
   {
      System.out.println("Default method");
   }

   public static void staticmethod()
   {
      System.out.println("Static method of Interface");
   }

   public static void staticmethod2(); //ERROR
}

class Maruti implements Car, Bike {
   public static void staticmethod()
   {
      System.out.println("Static method of Implementing class");
   }

}
```

```
public class InterfaceImplementationbasics1 {
    public static void main(String[] args) {
        Maruti.staticmethod();
        Car.staticmethod();

    }
}
```

OUTPUT:
Static method of Implementing class
Static method of Interface

Anonymous Inner Class

==========

->Can be used to provide body for abstract methods of interface or abstract class

Ex:

```
class Vehicle {
   void driver() {
      System.out.println("driving");
   }
}

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

      Vehicle v = new Vehicle() { //Anonymous Inner class
         @Override
         void driver() {
            System.out.println(" car driving");
         }
      };

      v.driver();


   }
}
```

OUTPUT:

car driving

->AIC can be used to provide body for all abstract methods of abstract class and interface while creating object itself.

Ex:

```
interface I {
   void add();
   void sub();
}

public class AnonymousInnerClass2 {
   public static void main(String[] args) {
      I i = new I() {

         @Override
```

```
        public void add() {
            System.out.println("adding");
        }

        @Override
        public void sub() {
            System.out.println("subtracting");
        }
    };

    i.add();
    i.sub();
  }
}
```

OUTPUT:
adding
subtracting

Arrays

=======

->Object used to store collection of elements.
->Memory will be allocated in heap area
->Can store primitive type of data and also object type of data[ref values of class]
->Index based data structure to store large volume of homogeneous(similar) type of data
->Gives size of an array a.length()
->Data type can be primitive or Object type
->To get the class name: a.getClass().getName()

->Other syntax
String s[]=new String[4];
String []s=new String[]{"appu","shivu"};
int [][]a={{1,2,3},{4,5,6}};

Syntax:
datatype [] refVar=new datatype[size];
Ex:
int a[]=new int[5];
a[0]=10;
a[2]=20;

sop(a[1]); //0 default value

a->[10,0,20,0,0,0]

Ex: int[] a={1,2,3,4};

```
public class OneDArray {
   public static void main(String[] args) {
      int a[]=new int[5];
      Scanner sc=new Scanner(System.in);
      for (int i=0;i<a.length;i++)
      {
         System.out.println("Enter input");
         a[i]=sc.nextInt();
      }

      System.out.println("Size is: "+a.length);

      for(int i:a)
      {
         System.out.print("Value is: "+i+" ");
      }
   }
```

}

2D Array

---

5 students in 3 different classroom

class 0 s0 s1 s2 s3 s4
class 1 s0 s1 s2 s3 s4
class 2 s0 s1 s2 s3 s4

int [][]a=new int[3][5];

Ex:

```
public class TwoDArray {
   public static void main(String[] args) {
      int[][] a = new int[3][5];

      Scanner sc = new Scanner(System.in);
      for (int i = 0; i < a.length; i++) {
         System.out.println("Enter input");
         for (int j = 0; j < a[i].length; j++) {
            a[i][j] = sc.nextInt();
         }
      }
      System.out.println("Size is: " + a[1].length);

      for (int i[] : a) {
         for (int j : i) {
            System.out.print(j + " ");
         }
         System.out.println();
      }

   }
}
```

OUTPUT:
Size is: 5
1 22 2 33 3
44 5 66 7 77
6 55 44 4 33

Jagged Array

2D Jagged Array
class 0 s0 s1 s2
class 1 s0 s1 s2 s3
class 2 s0 s1 s2 s3 s4

Ex:

```
package oopstelusko;

public class JaggedArray {
   public static void main(String[] args) {
      int [][] a=new int[3][];
      a[0]=new int[3];
      a[1]=new int[4];
      a[2]=new int[5];

      a[0][0]=10;
      a[0][1]=20;
      a[0][2]=30;

      a[1][0]=10;
      a[1][1]=20;
      a[1][2]=30;
      a[1][3]=40;

      a[2][0]=10;
      a[2][1]=20;
      a[2][2]=30;
      a[2][3]=40;
      a[2][4]=50;

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

   }
}
```

OUTPUT:
10 20 30
10 20 30 40
10 20 30 40 50

Ex: Object type Array

```
package oopstelusko;

class Student {
   String name;
   int age;

   public Student(String name, int age) {
      this.age = age;
      this.name = name;
   }

   public void display() {
      System.out.println(name + " age is " + age);
   }
}

public class StudentClassArray {

   public static void main(String[] args) {
      Student st[] = new Student[3];
      Student s1 = new Student("appu1", 22);
      Student s2 = new Student("appu2", 12);
      Student s3 = new Student("appu3", 32);


      st[0] = s1;
      st[1] = s2;
      st[2] = s3;

      for (Student s : st) {
         s.display();
         System.out.println("********");
      }

   }


}
```

OUTPUT:
appu1 age is 22
****
appu2 age is 12
****
appu3 age is 32
****

Limitation

->Size is fixed and need continuous memory location.
->Can store only same type of data
->No direct class is there to work with data or array elements.

Arrays Class

---

int a[]={2,1,3,4,2}'
Arrays.sort(a); //changes in mutable way, modify in original content

String Class

==========

->Anything inside " " is a String object.

Ex: "123" "a" " " "appu"

Types

Mutable: Changable Ex: StringBuffer and StringBuilder

Immutable: Non Changable Ex: String class objects

String

->Heap[Allow duplicates]/SCP[Present inside Heap and not allow duplicates]

->If we use new keyword to create string data, then data will be created inside heap and scp. Reference variable will point to heap area. JVM internal reference will points to SCP

->If we not use new keyword to create string data, then data will be created inside SCP. Reference variable will point to SCP.

->

String str1="appu";

Created inside SCP

String str1=new String("appu");

Created in heap and scp

->

== : reference value comparison

equals(): content comparison

->Ex:

Case 1:

String s1="appu";

String s2="appu";

s1 100

  appu [SCP]

s2 100

both s1 and s2 reference variable points to same object inside SCP with same reference value 100

Case 2:

String s1=new String("appu");

String s2=new String("appu");

s1 100 appu[Heap1] and [SCP]

s2 200 appu[Heap2] and [SCP]

s1 points to Heap1 object and having 100 as reference value
s2 points to heap2 and having 200 as reference value
one copy is also created inside SCP and JVM implicitly points to it.[not ref variable]

->GC cannot delete data inside SCP, it will be cleared once JVM is shutdown

String concatenation

->Using concat() or using + operator
->Combining two string and store new string in new object
->But when we concat, new object will be created and new modified content will be stored there and original data will not be modified in original object.

```
Strig s1=new String("appu");
s1.concat("raj");
sop(s1); //appu
String s1=s1.concat("raj");
sop(s1); //appu raj
```

Ex:

```
String s="appu"+100+33; //appu10033
String s=12+2+"appu"; //14appu
```

In Built Methods

---

equals()
equalIgnoreCase()
toLowerCase()
toUpperCase()
charAt(int index)
substring(int startIndex,int endingIndex)
substring(int startIndex)
compareTo()
concat()
contains(String str)
index("string str") //first occurrence
lastIdexOf("string str") //last occurrence
length()
startsWith("String str")
endsWith("String str")
toCharArray()
split("String str")

toString()

String Buffer and String Builder

->Mutable version of the String object
->Any modification will be done in same object.

```
StringBuffer sb=new StringBuffer("appu");
sop(sb); //appu
sb.append("raj");
sop(sb); //appuraj
```

->String Buffer: synchronized methods[not supports multithreading]
String Builder: non synchronized methods[supports multithreading]

->

```
StringBuffer sb1=new StringBuffer("appu");
StringBuffer sb2=new StringBuffer("appu");
sop(sb1.equals(sb2)); //false , equals method of object class will be called and it will do reference value comparison not content comparison.
but in String class equals() of Object class is overridden so it do content comparison
sop(sb1.compareTo(sb2));
```

```
->final StringBuffer sb=new StringBuffer("appu");
sb.append("raj");
sop(sb); //appuraj
```

->final dont have any impact on mutability of String buffer object. It impact only reference value not the content.

Interface and Lambda Expression

==========

Interface Types

Normal Interface

SAM/Functional Interface: Has single abstract method Ex: Runnable Interface

Marker Interface: Has 0 abstract method

Ex: Serializable Interface

SAM or Functional Interface

->Interface having single abstract method , can have any number of default, static, private methods.

Lambda Expression

->Used to provide body for abstract method of SAM interface.

->Not create extra class files

Rules:

argument within () if there are two or more, else we can remove ()

need to add ->

single statement no need of {}

return needs { }

end ;

Ex:

```
I2 i = (a, b) ->
      {
         return a + b;
      };
```

Ex 1:

```
interface I1 {
   void show();
}

public class Lambda1 {
   public static void main(String[] args) {

//Anonymous Inner Class implementation for Interface
//       I1 i=new I1()
//       {
//          @Override
```

```
//          public void show() {
//              System.out.println("in I1 show");
//          }
//      };

//Lambda implementation for Interface
      I1 i = () ->
      {
         System.out.println("in I1 show");
      };

      i.show();
   }
}
```

OUTPUT:
in I1 show

Ex 2:

```
interface I2 {
   int add(int a, int b);
}

public class Lambda2 {
   public static void main(String[] args) {
      I2 i = (a, b) ->
      {
         return a + b;
      };
      System.out.println("Sum is: " + i.add(10, 23));
   }
}
```

Ex 3:

```
interface I3 {
   int findLength(String str);
}

public class Lambda3 {
   public static void main(String[] args) {
      I3 i = (str) ->
      {
         int len = str.length();
```

```
        System.out.println("String is: " + str);
        return len;
    };

    System.out.println("Length of the String is: " + i.findLength("appu is our boss god is idiot"));
  }
}
```

->Lambda exp for forEach method of List interface

Ex:

```
public class ForEachDemo {
  public static void main(String[] args) {
    List<Integer> list=new ArrayList<Integer>();
    list.add(10);
    list.add(20);
    list.add(30);
    list.forEach(n-> System.out.println(n));
  }
}
```

Default method

->Method with body inside Interface

Exception handling

==========

->Exception is an mistake occurs at run time that may cause abnormal termination of the application.
->Handling such a exception is knows as Exception handling

Case 1:
Exception occurs in main method
Exception Object is created and throws to JVM
JVM checks if there is as Exception handler created by developer ie respective catch block in main method
If Yes
 JVM throws to User defined exception handler created by developer ie catch block to handle the exception
 Smooth termination of the program, continues from where it stopped.
If No
 JVM throws it to Default exception handler
 Abrupt termination of the program where exception occurred.

Ex: Without exception handler

```
public class Exceptionhandler1 {
   public static void main(String[] args) {
      System.out.println("Connection Started");
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numerator and Denominator");
      int num = sc.nextInt();
      int den = sc.nextInt();
      int result = num / den;
      System.out.println("Result is: " + result);
      System.out.println("Connection Ended");
   }
}
```

OUTPUT:

```
Enter Numerator and Denominator
22
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exceptionhandlingtelusko.Exceptionhandler1.main(Exceptionhandler1.java:14)
```

Handling Exception Steps

Identify risky code where exception may occur and put in to try block
In catch block handle it
Whenever exception occurs in try block statement, control comes to catch block and execute it.
The statement below the exception occurred statement will not be executed.

Solution: With Exception handler

```
public class Exceptionhandler1 {
   public static void main(String[] args) {
      System.out.println("Connection Started");
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numerator and Denominator");
      int num = sc.nextInt();
      int den = sc.nextInt();

      try {
         int result = num / den;
         System.out.println("Result is: " + result);
      } catch (Exception e) {
         System.out.println(e);
      }

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

OUTPUT:

```
Enter Numerator and Denominator
22
0
java.lang.ArithmeticException: / by zero
Connection Ended
```

Points

->Exception can be handled by try and catch block. Keep exception causing statements in try block and exception handling code in catch block.

Ex:

```
try {
         int result = num / den;
         System.out.println("Result is: " + result);
      } catch (Exception e) {
         System.out.println(e);
      }
```

->One try can have 0 or more catch block

Which type of exception is thrown by try block its respective catch block will be executed.

Ex:

```
try
{
```

```
int result = num / den;
}
catch(ArithmeticException e)
{
sop("Please enter valid data");
}
catch(NegativeArraySizeException e)
{
sop("Please enter valid data");
}
catch(ArrayIndexOutOfBoundException e)
{
sop("Please enter valid data");
}
catch(Exception e)
{
sop("Please enter valid data");
}
```

->Generic exception should be in last catch block.

->One try must follow with at least one catch or finally block.

->If exception occurs, then statement below try block will not be executed.

->Nested try catch block is allowed

Case 2:
Main method calls alpha()
Exception occurs in alpha()
Exception Object is created and throws to JVM
JVM checks if there is as Exception handler created by developer ie respective catch block in alpha()
If Yes
JVM throws to User defined exception handler created by developer ie catch block in alpha() to handle the exception
Smooth termination of the program, continues from where it stopped.
If No
JVM Checks if there is as Exception handler created by developer ie respective catch block in main()
If Yes
JVM throws to User defined exception handler created by developer ie catch block in main() to handle the exception
Smooth termination of the program, continues from main().
Statement after exception causing statement inside alpha() will not be executed , only statement present inside main() after calling alpha() will be executed.

If No

JVM throws it to Default exception handler

Abrupt termination of the program where exception occurred. All the statements after exception causing statement inside alpha() will not be executed and all the statement after calling alpha() inside main() will not be executed.

Ex: Without exception handler in both main() and alpha()

```
package exceptionhandlingtelusko;

import java.util.Scanner;

class Sample {
   void alpha() {
      System.out.println("Alpha Started");
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numerator and Denominator");
      int num = sc.nextInt();
      int den = sc.nextInt();
      int result = num / den;
      System.out.println("Result is: " + result);
      System.out.println("Alpha Ended");
   }
}

public class CallingAndCalledmethodExceptionhandling {
   public static void main(String[] args) {
      System.out.println("Main Started");
      Sample s = new Sample();
      s.alpha();
      System.out.println("Main Ended");
   }
}
```

OUTPUT:

```
Main Started
Alpha Started
Enter Numerator and Denominator
22
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at
exceptionhandlingtelusko.Sample.alpha(CallingAndCalledmethodExceptionhandling.java:12)
```

at
exceptionhandlingtelusko.CallingAndCalledmethodExceptionhandling.main(CallingAndCalledmethod
Exceptionhandling.java:22)

Ex: Exception handler inside main()

```
package exceptionhandlingtelusko;

import java.util.Scanner;

class Sample {
   void alpha() {
      System.out.println("Alpha Started");
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numerator and Denominator");
      int num = sc.nextInt();
      int den = sc.nextInt();
      int result = num / den;
      System.out.println("Result is: " + result);
      System.out.println("Alpha Ended");
   }
}

public class CallingAndCalledmethodExceptionhandling {
   public static void main(String[] args) {
      System.out.println("Main Started");
      Sample s = new Sample();
      try
      {
         s.alpha();
      }
      catch (Exception e)
      {
         System.out.println("Invalid Input");
      }

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

OUTPUT:

```
Main Started
Alpha Started
Enter Numerator and Denominator
22
```

0
Invalid Input
Main Ended

Ex: Exception handler inside alpha()

```
package exceptionhandlingtelusko;

import java.util.Scanner;

class Sample {
   void alpha() {
      System.out.println("Alpha Started");
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numerator and Denominator");
      int num = sc.nextInt();
      int den = sc.nextInt();
      try {
         int result = num / den;
         System.out.println("Result is: " + result);
      } catch (Exception e) {
         System.out.println("Invalid Input");
      }
      System.out.println("Alpha Ended");
   }
}

public class CallingAndCalledmethodExceptionhandling {
   public static void main(String[] args) {
      System.out.println("Main Started");
      Sample s = new Sample();
      s.alpha();
      System.out.println("Main Ended");
   }
}
```

OUTPUT:

```
Main Started
Alpha Started
Enter Numerator and Denominator
22
0
Invalid Input
Alpha Ended
Main Ended
```

Ex: Exception handler in both main() and alpha()

```
package exceptionhandlingtelusko;

import java.util.Scanner;

class Sample {
   void alpha() {
      System.out.println("Alpha Started");
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numerator and Denominator");
      int num = sc.nextInt();
      int den = sc.nextInt();
      try {
         int result = num / den;
         System.out.println("Result is: " + result);
      } catch (Exception e) {
         System.out.println("Invalid Input");
      }
      System.out.println("Alpha Ended");
   }
}

public class CallingAndCalledmethodExceptionhandling {
   public static void main(String[] args) {
      System.out.println("Main Started");
      Sample s = new Sample();
      try
      {
         s.alpha();
      }catch (Exception e) {
         System.out.println("Invalid Input");
      }

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

OUTPUT:

```
Main Started
Alpha Started
Enter Numerator and Denominator
22
0
```

Invalid Input
Alpha Ended
Main Ended

Exception Information

->
e.getMessage(): Prints the description of the exception
Ex: /by zero

e.toString(): Prints name and description of the exception
Ex: Arithmetic Exception:/by zero

e.printStackTrace(): Prints the name and the description of the Exception along with stack trace

Exception handling cases

->Whenever exception generated we can do three things
  case 1: handle exception using try and catch
  case 2: ducking[bypass] the exception using throws keyword
  case 3: rethrowing the exception using throw,throws,try,catch,finally

case 2: ducking[bypass] the exception from called to caller using throws keyword

If the exception occurring method cannot able to handle the exception then it can inform the caller method to handle it using throws keyword.

Ex: called->caller Ex: alpha()->main()
main() handles it

```
class Sample1 {
   void alpha() throws Exception{
      System.out.println("Alpha Started");
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numerator and Denominator");
      int num = sc.nextInt();
      int den = sc.nextInt();
      int result = num / den;
      System.out.println("Result is: " + result);
      System.out.println("Alpha Ended");
   }
}
```

```
public class ThrowsDemo {
```

```
    public static void main(String[] args) {
        System.out.println("Main Started");
        Sample1 s = new Sample1();
        try {
            s.alpha();
        } catch (Exception e) {
            System.out.println("Invalid Input");
        }

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

OUTPUT:
Main Started
Alpha Started
Enter Numerator and Denominator
22
0
Invalid Inputo JVM to han
Main Ended

Ex:
called->caller Ex: alpha()->main()
called->JVM Ex: main()->JVM

Both alpha and main() by pass the exception and it finally reaches the JVM Default exception handler to handles it

```
package exceptionhandlingtelusko;

import java.util.Scanner;

class Sample1 {
    void alpha() throws Exception {
        System.out.println("Alpha Started");
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Numerator and Denominator");
        int num = sc.nextInt();
        int den = sc.nextInt();
        int result = num / den;
        System.out.println("Result is: " + result);
        System.out.println("Alpha Ended");
    }
}
```

```
public class ThrowsDemo {
   public static void main(String[] args) throws Exception{
      System.out.println("Main Started");
      Sample1 s = new Sample1();
      s.alpha();
      System.out.println("Main Ended");
   }
}
```

OUTPUT:
Main Started
Alpha Started
Enter Numerator and Denominator
22
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at exceptionhandlingtelusko.Sample1.alpha(ThrowsDemo.java:12)
	at exceptionhandlingtelusko.ThrowsDemo.main(ThrowsDemo.java:23)

Types of Exception

->**Checked and Un Checked Exception**
Checked Exception: Exception identified at compilation time.
Ex:IOException, InterruptedException, SQLException
Un Checked Exception: Exception identified by JVM
Ex:RuntimeException, ArithmeticException, ClassCastException

Finally block

->Follows try or catch block, whether exception occur or not finally block will be executed.
->For clean up resource we can use it along with try and catch
->try or catch execute or not, finally will be executed.
->finally can also contain try catch inside it
->Exception occurs, Handled then smooth termination, else abnormal termination. But in either case finally will be executed.
->System.exit(0) dominates finally block.

Ex:
try
{
open file

```
read data[chances of exception]
close file
}
catch()
{
...
}
```

If exception occurs close file statements wont be executed.

Solution:

```
try
{
open file
read data[chances of exception]
}
catch()
{
...
}
finally
{
close file
}
```

Ex:

```
class Sample2 {
   void alpha() {
      System.out.println("Alpha Started");
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numerator and Denominator");
      int num = sc.nextInt();
      int den = sc.nextInt();
      try {
         int result = num / den;
         System.out.println("Result is: " + result);
      } catch (Exception e) {
         System.out.println("Invalid Input");
         throw e; //control moves to main()
      } finally {
         System.out.println("Alpha Ended");
      }
   }
```

```
}

public class ThrowKeyword1 {
   public static void main(String[] args) {
      System.out.println("Main Started");
      Sample2 s = new Sample2();
      try {
         s.alpha();
      } catch (Exception e) {
         System.out.println("IHey something went wrong");
      }
      System.out.println("Main Ended");
   }
}
```

OUTPUT:

```
Main Started
Alpha Started
Enter Numerator and Denominator
22
0
Invalid Input
Alpha Ended
IHey something went wrong
Main Ended
```

->finally vs return: finally will dominate return statement.

Ex:

```
class Calc {
   int add() {
      try {
         return 100;
      } catch (Exception e) {
         return 200;
      } finally {
         return 300;
      }
   }
}

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

```
        Calc c = new Calc();
        System.out.println(c.add());
    }
}
```

OUTPUT:
300

Overview

---

try
To keep risky code

catch
to handle exception

finally
to keep cleanup code

throw
to rethrow the exception manually

throws
to by pass the exception from called to caller method

Exception Hierarchy

Object
  Throwable
    Error
    Exception

Error
  VirtualMachineError
    OutOfMemoryError
     StackOverflowError
  LinkageError
  AssertionError

Exception
  RuntimeException
  IOException
  InterruptedException
  SQLException

RuntimeException
 ArithmeticException
 NullPointerException
 ClassCastException
 IndexOutOfBoundException [ArrayIdexOutOfBound/StringIndexOutOfBoundException]
 NegativeArraySizeException

IOException
  EOFException
  FileNotFoundException
  RemoteException

Runtime Error

Method calling itself. Ex: StackOverflowError
LinkageError
AssertionError

throw keyword

->Used to handle and re throw the exception

```
try
{..}
catch(Excption e)
{
..... //handled
throw e; //Throws an exception after handling to caller method
}
```

Ex:

```
package exceptionhandlingtelusko;

import java.util.Scanner;

class Sample2 {
   void alpha() {
      System.out.println("Alpha Started");
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numerator and Denominator");
      int num = sc.nextInt();
      int den = sc.nextInt();
      try {
         int result = num / den;
```

```
            System.out.println("Result is: " + result);
        } catch (Exception e) {
            System.out.println("Invalid Input");
            throw e;
        }
        System.out.println("Alpha Ended");
    }
}

public class ThrowKeyword1 {
    public static void main(String[] args) {
        System.out.println("Main Started");
        Sample2 s = new Sample2();
        try {
            s.alpha();
        } catch (Exception e) {
            System.out.println("Hey something went wrong");
        }
        System.out.println("Main Ended");
    }
}
```

OUTPUT:
Main Started
Alpha Started
Enter Numerator and Denominator
22
0
Invalid Input
Hey something went wrong
Main Ended

->To throw the exception manually we use throw keyword

Ex: throw manually system defined exception

```
class Launch
{
main()
{
throw new ArithmeticException("/ by zero");
}
```

Ex:

```
public class Throw2 {
```

```
    public static void main(String[] args) {
        System.out.println("Start");
        try
        {
            throw new ArithmeticException("/ by zero");
        }
        finally
        {
            System.out.println("End");
        }
    }
}
```

OUTPUT:

```
Start
End
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exceptionhandlingtelusko.Throw2.main(Throw2.java:8)
```

Custom Exception

Ex:

```
class AccountBlocked extends Exception
{
    AccountBlocked(String str)
    {
        super(str);
    }
}

class Bank{
    int pin=1234;

    void validatePin() throws AccountBlocked {
        System.out.println("Enter Pin");
        Scanner sc=new Scanner(System.in);
        int enteredPin=sc.nextInt();
        if(pin==enteredPin)
        {
            System.out.println("Take Amount");
        }
        else
        {
            throw new AccountBlocked("Kalla");
        }
```

```
   }
}
public class BankApp {
   public static void main(String[] args) throws AccountBlocked {
      Bank b=new Bank();
      try
      {
         b.validatePin();
      }
      catch(AccountBlocked a)
      {
         System.out.println("Due to invalid PIN your account is blocked");
         System.out.println(a);
      }

   }
}
```

OUTPUT:
Enter Pin
123
Due to invalid PIN your account is blocked
exceptionhandlingtelusko.AccountBlocked: Kalla

Multiple Catch Block

->One try can have multiple catch block.

```
try
{...}
catch(XXX|XXX|XXX e)
{....}
```

Possible cases

---

only try or catch or finally not allowed
catch or finally should follow try

Packages

==========

->Folder contains classes of similar types.

->Collection of .class files

->Used for modularity

->User defined/Pre defined Ex: java.util.*

->One package can contains another sub package

->Compiler check package exist, JVM includes required classes of the package during runtime.

->If the classes present in same package then we can use directly else we need to import it before use.

```
import java.util.Scanner;
import java.util.*;
```

Collection

==========

->Interface to store homogeneous and heterogeneous type of data as single unit

->Interface used to store Objects of different classes

->If we want to store primitive type of data then we need to convert to object type using wrapper class and store it into collection.

methods

----

add()

contains()

clear()

indexof()

size()

Collection Hierarchy

Iterable

  Collection[I]

    List[I]

    Queue[I]

    Set[I]

List[I]

  ArrayList

  LinkedList

Queue[I]

  DeQueue[I]

    LinkedList

    ArrayDeQueue

  PriorityQueue

Set[I]

  SortedSet[I]

    TreeSet

  HashSet

     LinkedHashSet

ArrayList

Implements: List Interface
DS: Doubly linked list
Insertion order: Yes
Sorting order: No
Index based: Yes
Duplicates: Yes
Null values: Yes
Type of data: Homo and Heto
Items added: Only rear end
Suitable for: Insertion operation

Ex:

```
public class ArrayListDemo {
   public static void main(String[] args) {
      List<Integer> list=new ArrayList<Integer>();

      list.add(10);
      list.add(20);
      list.add(10);
      list.add(null);

      for(Integer i:list)
      {
         System.out.println("Data: "+i);
      }
   }
}
```

LinkedList

Implements: List Interface and DQueue Interface
DS: Doubly linked list
Insertion order: Yes
Sorting order: No
Index based: Yes
Duplicates: Yes
Null values: Yes
Type of data: Homo and Heto
Items added: Both Rear and front end
Suitable for:insertion operation

ArrayDQueue

Implements: DQueue interface
DS: Double ended Q
Insertion order: Yes
Sorting order: No
Index based: No
Duplicates: Yes
Null values: No
Type of data: Homo and Heto
Items added:Only rear end
Suitable for: Rear and Front end operation

PriorityQueue

Implements: Queue interface
DS: Minimum heap
Insertion order: No
Sorting order: No
Index based: No
Duplicates: Yes
Null values: No
Type of data: Homo
Items added: Only rear end
Suitable for: Rear and Front end operation

TreeSet

Implements: SortedSet interface
DS: Binary Search tree
Insertion order: No
Sorting order: Yes
Index based: No
Duplicates: No, If added it will remove inetrnally
Null values: No
Type of data: Homo [classcast exception if added]
Items added:
Suitable for: Search operation

HashSet

Implements: Set interface
DS: Hashing algo
Insertion order: No
Sorting order: No
Index based: No
Duplicates: No, If added it will remove inetrnally
Null values: Yes
Type of data: Homo and heto
Items added:
Suitable for: Search operation

LinkedHashSet: Same as HashSet, sub class of hashSet. Order of insertion is preserved.

Downcasting in Collection

->Storing Object type data into Integer type variable

```
public class DowncastingInCollection {
   public static void main(String[] args) {
      List list = new ArrayList();
      list = Arrays.asList(12, 23, 45, 66);

      Object obj=list.get(2);
      System.out.println(obj);

      Integer in=(Integer)list.get(2); //Downcasting
      System.out.println(in);

   }
}
```

For VS For Each VS Iterator

```
package collection;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

public class IteratorDemo {
   public static void main(String[] args) {
      List list = new ArrayList();
      list = Arrays.asList(12, 23, 45, 66);

      for (int i = 0; i < list.size(); i++) {
         System.out.print(list.get(i) + " ");
      }

      for (Object obj : list) {
         System.out.print(obj + " ");
      }

      Iterator it = list.iterator();
      while (it.hasNext()) {
         System.out.print(it.next() + " ");
      }
   }
}
```

OUTPUT:

```
***FOR LOOP RETRIEVE*******
Data is:
12 23 45 66
***FOR EACH LOOP RETRIEVE*******
Data is:
12 23 45 66
***ITERATOR RETRIEVE*******
Data is:
12 23 45 66
```

List Iterator

->To iterate in both front and reverse direction

Ex:

```
public class ListIteratorDemo {
   public static void main(String[] args) {
      List list = new ArrayList();
      list = Arrays.asList(12, 23, 45, 66);

      ListIterator lit = list.listIterator(list.size());

      while (lit.hasPrevious()) {
         System.out.print(lit.previous() + " ");
      }

      while (lit.hasNext()) {
         System.out.print(lit.next() + " ");
      }
   }
}
```

OUTPUT:

66 45 23 12

12 23 45 66

Collections Class

->Class contains static utility methods used to perform different operation on Collection objects

->Mutable way, original content will be modified

->Other methods: shuffle(), binarySearch(),sort(), ...

Ex:

```
public class ListIteratorDemo {
   public static void main(String[] args) {
      List list = new ArrayList();
      list = Arrays.asList(122, 23, 5, 66,0);

      Collections.sort(list);
      for(Object obj:list)
      {
         System.out.print(obj+" ");
      }
```

```
    }
}
```

OUTPUT:

```
0 5 23 66 122
```

Generics

=========

->To bring type safety

->Used to avoid ClassCast exception

->Generics can be same type or parent type.

->If class is declared as generics <T> type then it can accept any type of data.

List<Integer> list=new ArrayList<Integer>();

//Used to store only integer type of data

List<Object> list=new ArrayList<Object>(); //Can be used to store Integer type data, Object is parent of Integer

Map

=====

->Interface used to store data in form of key value pair

Key Value -> Entry
Object Object
Unique Can be duplicate

->Key must be unique and Value can be duplicate
->Type of data[key or value] can be anything
->key value pair is known as entry

Hierarchy

Map[I]
  HashTable
  HashMap
    LinkHashMap
  SortedMap[I]
    NavigableMap[I]
      TreeMap

LinkedHashMap: Preserves insertion order
TreeMap: Sorted order based on key
HashTable: null is not allowed in key and not supports multithreading
WeakHashMap: similar to HashMap, but GC dominates WeakHashMap while doing GC

HashMap

->Key: Unique[If duplicate then it will be override with new value]/One null allowed/Homo or Heto
value: Duplicate allowed/null allowed/Homo or Heto
->Not Maintains insertion order, if needs insertion order then go for LinkedHashMap if need sorted order based on key then go for TreeMap
->Supports multithreading

Ex:

```
public class HashMapDemo1 {
   public static void main(String[] args) {
      Map<Object, Object> map = new HashMap<>();
      map.put(1, "appu");
      map.put('a', 22);
      map.put(null, "shivu");

      System.out.println(map.size());
```

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

        Set set = map.keySet();
        for (Object obj : set) {
           System.out.println("Key is: " + obj);
        }

        Collection c = map.values();
        for (Object obj : c) {
           System.out.println("Value is: " + obj);
        }

        Set s = map.entrySet();
        for (Object obj : s) {
           System.out.println("Key value is: " + obj);
        }

        Set s1 = map.entrySet();
        Iterator it = s1.iterator();
        while (it.hasNext()) {
           Map.Entry pair =(Map.Entry) it.next();
           System.out.println(pair.getKey()+" value is "+pair.getValue()) ;
        }
     }
}
```

OUTPUT:

```
3

{null=shivu, 1=appu, a=22}

Key is: null
Key is: 1
Key is: a

Value is: shivu
Value is: appu
Value is: 22

Key value is: null=shivu
Key value is: 1=appu
Key value is: a=22

null value is shivu
```

1 value is appu
a value is 22

Points:
->Map contains Key Value pair
->Key can be of any type, Value can be of any type
->Combination of Key and Value is known as Entry , this combination is of Type Map.Entry [Entry is an interface present inside Map interface]
->Suppose Map contains 5 KeY Value pair then it contains Set of 5 Entry which is of type Map.Entry

Set set=map.entrySet(); //Each element in the set is one Map.Entry

Ex:

```
package collection;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

class Employee {
   String name;
   double salary;

   public Employee(String name, double salary) {
      this.name = name;
      this.salary = salary;
   }

   public String toString() {
      return name + " " + salary;
   }
}

public class HashMapDemo {
   public static void main(String[] args) {
      Employee e1 = new Employee("appu", 1234);
      Employee e2 = new Employee("appu", 1234);
      Employee e3 = new Employee("appu", 1234);

      Map map = new HashMap();
      map.put(1, e1);
      map.put(2, e2);
      map.put(3, e3);
```

```
        Set<Map.Entry> set = map.entrySet();

        Iterator it = set.iterator();
        while (it.hasNext()) {
            Map.Entry pair = (Map.Entry) it.next();
            System.out.println(pair.getKey() + ":" + pair.getValue());
        }

    }
}
```

OUTPUT:
1:appu 1234.0
2:appu 1234.0
3:appu 1234.0

Linked HashMap

->Same as HashMap but it maintains insertion order.

TreeMap

->Same as HashMap, but maintains sorted order based on key.
->Key should be of same type, else ClassCast exception
->Null not allowed

finalize()

->GC Before doing cleaning up of any Object it will execute finalize()

```
class Dummy {
    @Override
    public void finalize() {
        System.out.println("I am Finalize");
    }
}

public class FinalizeMethod {
    public static void main(String[] args) {
        Dummy d = new Dummy();
        d = null;
        System.gc();

    }
```

}

ForEach

->Predefined function for List interface to print the value

Ex:

```
public class ForEachDemo {
   public static void main(String[] args) {
      List list = new ArrayList();

      list = Arrays.asList(12, 32, 44, 1, 2, 5, 6666);
      list.forEach(n -> System.out.println(n));
   }
}
```

Stream API

==========

->Used to process the data present in the collection.

->When we do any modification on old stream [filter or map] then existing stream will go away and new stream will be created.

->Once used then its finished cant use again, it throws exception

Ex:

```
List<Integer> list = new ArrayList<Integer>();
list = Arrays.asList(23, 11, 2,21,20,0, 89, 6);
Stream<Integer> s1 = list.stream();

s1.forEach(s-> System.out.print(s+" ")); //used once
s1.count(); //exception because s1 already once used java.lang.IllegalStateException
```

Solution: combine both and use or create new stream and use

```
s1.forEach(s-> System.out.print(s+" "));
s2.count();
```

Ex: Convert List into Stream and print its value

```
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<>();
      list = Arrays.asList(23, 11, 2, 0, 89, 6);
      Stream s1 = list.stream();
      s1.forEach(x -> System.out.println(x));
   }
```

->Once we perform any operation then its end we cant re use same again. Need to store into new Stream and use it.

Filter: To filter the data present inside the collection based on some condition
Map: To process data present inside the collection based on some logic.
Reduce: To merge the data

Collection->Streams : Bring data present in the collection into Streams
Apply operation[Filter/Map/Reduce] on the data present in the Streams.

stream() : Default method

->Once operation is performed the stream will be expired.

Map()

->Used to modify the data present in the Stream.
->Take each element from stream and perform the operation[calls Function apply()] and update new data into the same or new stream.
->Takes Function as a argument and return Stream as output

Function:
->Takes 2 argument
Input: any type
Return value: any type.
Output: It returns any type modified value based on logic.

Ex: Function takes Integer as input and return Integer as output.

```
    input type return type
Function<Integer,Integer> f=new Function<Integer,Integer>()
{

public Integer apply(Integer x) //apply is a method inside Function interface which accepts arg and return value
{
//any logic
return x*x;
}
```

Ex:

```
public class Map1 {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<>();
      list = Arrays.asList(23, 11, 2, 0, 89, 6);
```

```
        Stream s1 = list.stream();

        //Need to double the value
        Function<Integer,Integer> function=new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer integer) {
                return integer*integer;
            }
        };

        //Map accepting function as arguement
        s1.map(function).forEach(x-> System.out.print(x+" "));
    }
}
```

OR Using lambda with Function

```
public class MapWithLambda {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list = Arrays.asList(23, 11, 2, 0, 89, 6);
        Stream s1 = list.stream();

        //Need to double the value
        Function<Integer, Integer> function = integer -> {
            return integer * integer;
        };
        //Map accepting function as arguement
        s1.map(function).forEach(x -> System.out.print(x + " "));
    }
}
```

OR Using Lambda directly

```
public class MapWithLambda {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list = Arrays.asList(23, 11, 2, 0, 89, 6);
        Stream<Integer> s1 = list.stream();
        s1.map(n -> n * n).forEach(x -> System.out.print(x + " "));
    }
}
```

OUTPUT:
529 121 4 0 7921 36

Filter

->Used to filter data present in the stream based on some condition. Take each element from Stream and check the condition[Calls Predicate test()] and matching condition data will be put back to Stream.
->Takes Predicate as Input and return Stream as Output

Predicate:
Input: any type data
Return value: boolean
Output: It returns true or false based on condition

Ex:

```
Predicate<Integer> p=new Predicate<Integer>() {
        @Override
        public boolean test(Integer integer) {
           return integer>20;
        }
     };
```

Ex: Filter with Predicate

```
public class FilterWithPredicate {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list = Arrays.asList(23, 11, 2, 0, 89, 6);
      Stream<Integer> s1 = list.stream();

      Predicate<Integer> p=new Predicate<Integer>() {
         @Override
         public boolean test(Integer integer) {
            return integer>20;
         }
      };

      s1.filter(p).forEach(x-> System.out.print(x+" "));
   }
}
```

OR

Ex: Filter with Predicate Lambda

```
public class FilterDemo1 {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list = Arrays.asList(23, 11, 2, 0, 89, 6);
      Stream<Integer> s1 = list.stream();
      s1.filter(x->x>20).forEach(x-> System.out.print(x+" "));
   }
}
```

OUTPUT:
23,89

Ex: Combining Map and Filter

```
public class MapWithFilterDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list = Arrays.asList(23, 11, 2,21,20,0, 89, 6);
      Stream<Integer> s1 = list.stream();
      s1.sorted().filter(x -> x >= 20).map(x -> x * x).forEach(x -> System.out.print(x + " "));
   }
}
```

OUTPUT:
400 441 529 7921

Reduce

->Merge all the elements
->Take all the elements from streams and combine it based on specified logic and return single result.
Input: BinaryOperator class object
Output: Optional class object

Ex:

```
public class ReduceDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list = Arrays.asList(23, 11, 2,21,20,0, 89, 6);
      Stream<Integer> s1 = list.stream();

      BinaryOperator<Integer> b=new BinaryOperator<Integer>() {
         @Override
         public Integer apply(Integer integer, Integer integer2) {
```

```
            return integer+integer2;
          }
       };

       Optional o=s1.reduce(b);
       System.out.println(o.get());
    }
}
```

OUTPUT:
172

Terminal Function

->Any function which not returns stream are terminal fucntions.

foreach() : Iterate stream

Ex:

```
public class TerminalFunctionDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list = Arrays.asList(23, 11, 2,21,20,0, 89, 6);
        Stream<Integer> s1 = list.stream();

        s1.forEach(s-> System.out.print(s+" "));
    }
}
```

count() : count no of elements in the stream

return long

Ex:

```
public class TerminalFunctionDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list = Arrays.asList(23, 11, 2,21,20,0, 89, 6);
        Stream<Integer> s1 = list.stream();

        long l=s1.count();
        System.out.println(l);
    }
}
```

findFirst(): return first element from the stream
return Optional class object
use get() to get the value

Ex:

```
public class TerminalFunctionDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list = Arrays.asList(23, 11, 2,21,20,0, 89, 6);
      Stream<Integer> s1 = list.stream();

      Optional o=s1.findFirst();
      System.out.println(o.get());
   }
}
```

Ex:

```
public class TerminalFunctionDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list = Arrays.asList(23, 11, 2,21,20,0, 89, 6);
      Stream<Integer> s1 = list.stream();

      Optional o=s1.findFirst();
      if(o.isPresent())
      {
         System.out.println(o.get());
      }


   }
}
```

Ex:

```
public class TerminalFunctionDemo {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<Integer>();
      list = Arrays.asList(23, 11, 2,21,20,0, 89, 6);
      Stream<Integer> s1 = list.stream();

      Integer i=s1.findFirst().orElse(0);
      System.out.println(i);


   }
```

}

Similarly findAny()

Overview:

Collection->Streams
Filter-->Predicate-->Streams
Map-->Function-->Streams

Terminal functions:
reduce/count/foreach/min/max/findany/findfirst

Java 8 Other features

==========

Date

---

date d=new Date();
sop(d);

var

->No need to specify data type
->need to assign value during declaration only
->only for local variables

Ex:
var c="appu";
System.out.println(c);

var i=12;
sop(i);

var a=new int[2];

textblock

---

to add line space
String str="""
<html>
,head>
""";

Output:
<html>
<head>

Sealed

->To restrict the extending of class to specific class

Ex:
sealed class A permits B{}

non-sealed class B extends A{}

class D extends A{} //Error D cant extend sealed class A

```
public class SealedExtendDemo {
}
```

Switch

Ex:

Ex: Lambda and removing break

```
public class SwitchDemo1 {
   public static void main(String[] args) {
      String day = "fri";
      int alaramTime;
      switch (day) {
         case "mon", "tue" -> alaramTime = 5;
         case "wed", "thur" -> alaramTime = 6;
         case "fri" -> alaramTime = 7;
         default -> alaramTime = 9;
      }

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

->Case can have multiple values

->Lambda in switch, we can remove break

->Common variable can be used switch as expression

Normal:

```
public class SwitchDemo1 {
   public static void main(String[] args) {
      String day = "mon";
      int alaramTime;
      switch (day) {
         case "mon":
            alaramTime = 5;
            break;
         case "tue":
            alaramTime = 6;
            break;
         case "wed":
            alaramTime = 5;
            break;
```

```
            case "thur":
                alaramTime = 5;
                break;

            default:
                alaramTime = 9;

        }

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

Ex: Combining Case values

```
public class SwitchDemo1 {
    public static void main(String[] args) {
        String day = "fri";
        int alaramTime;
        switch (day) {
            case "mon", "tue":
                alaramTime = 5;
                break;

            case "wed", "thur":
                alaramTime = 6;
                break;

            case "fri":
                alaramTime = 7;
                break;

            default:
                alaramTime = 9;
        }

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

Ex: Lambda and removing break

```
public class SwitchDemo1 {
    public static void main(String[] args) {
        String day = "fri";
```

```
        int alaramTime;
        switch (day) {
            case "mon", "tue" -> alaramTime = 5;
            case "wed", "thur" -> alaramTime = 6;
            case "fri" -> alaramTime = 7;
            default -> alaramTime = 9;
        }

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

Ex: using switch as expression, removing common variable and make it as single

```
public class SwitchDemo1 {
    public static void main(String[] args) {
        String day = "fri";
        int alaramTime;
        alaramTime = switch (day) {
            case "mon", "tue" -> 5;
            case "wed", "thur" -> 6;
            case "fri" -> 7;
            default -> 9;
        };
        System.out.println(alaramTime);
    }
}
```

Ex: Remove -> and use yield

```
public class SwitchDemo1 {
    public static void main(String[] args) {
        String day = "fri";
        int alaramTime;
        alaramTime = switch (day) {
            case "mon", "tue" : yield 5;
            case "wed", "thur" : yield 6;
            case "fri" : yield 7;
            default : yield 9;
        };
        System.out.println(alaramTime);
    }
}
```