Core Java Master Course

OOPs

Data Structure

Modularity

Design patterns

Concurrency

JVM

Java New Features

Java Stream

Rajeev Gupta

rgupta.mtech@gmail.com

https://www.linkedin.com/in/rajeevguptajavatrainer

rgupta.mtech@gmail.com

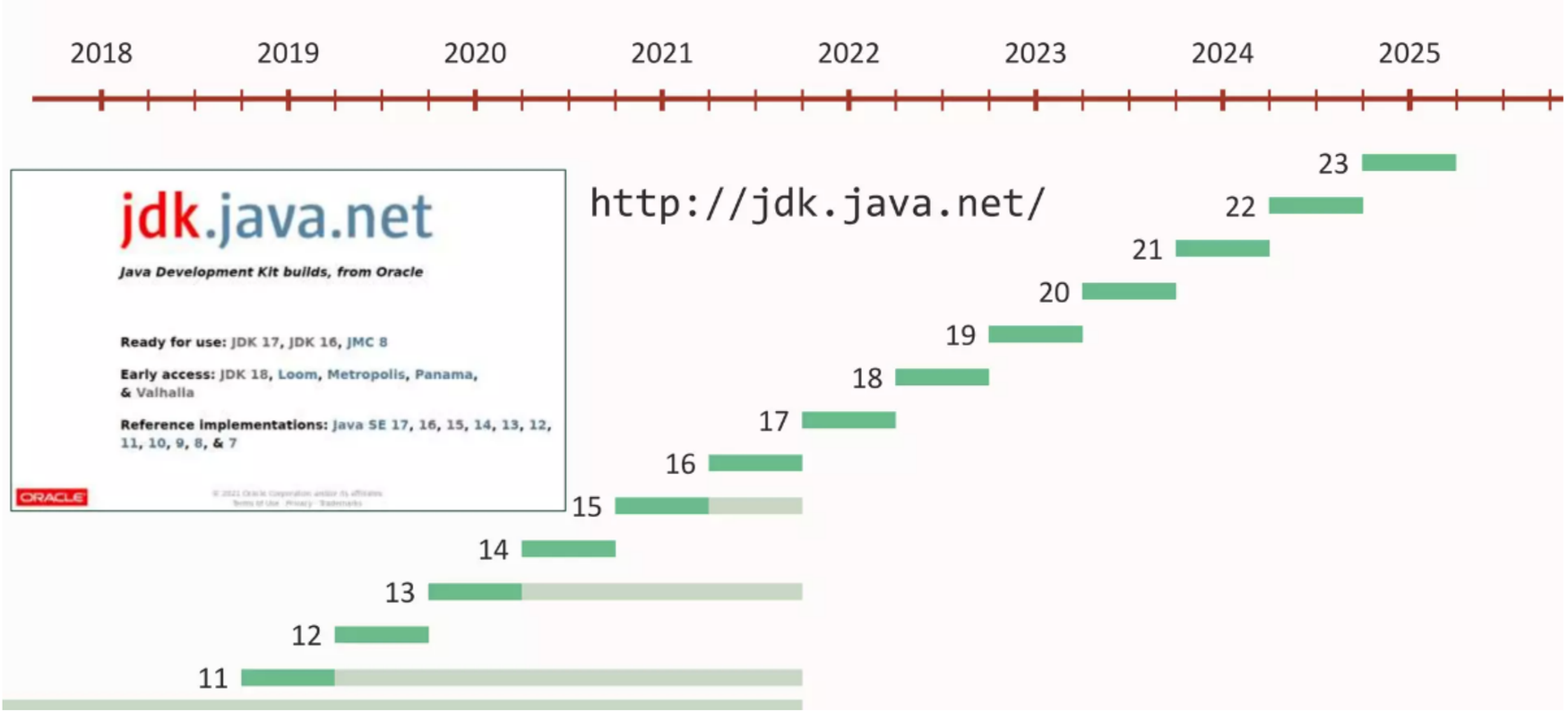
Session 1:
Object Orientation Fundamentals
Session 2:
Basic OOPs, Inheritance, Overriding, Overloading, Polymorphism
Session 3
Advance Object Orientation
Session 4:
String, Wrapper classes, Immutability
Inner classes, Java 5 features
Session 5:
Java Exception Handling, IO, Serilization
Session 6:
Java IO, Serilization
Session 7:
Java Collection API

rgupta.mtech@gmail.com

Session 1
Object Orientation Fundamentals

rgupta.mtech@gmail.com

Java Versions



rgupta.mtech@gmail.com

Java Versions

Version	Release date	End of Public Updates[5]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	January 2019 (commercial) December 2020 (non-commercial)	March 2025
Java SE 9	September 2017	March 2018	N/A
Java SE 10 (18.3)	March 2018	September 2018	N/A
Java SE 11 (18.9 LTS)	September 2018	March 2019 from Oracle Later from OpenJDK	Vendor specific
Java SE 12 (19.3)	March 2019	September 2019	N/A

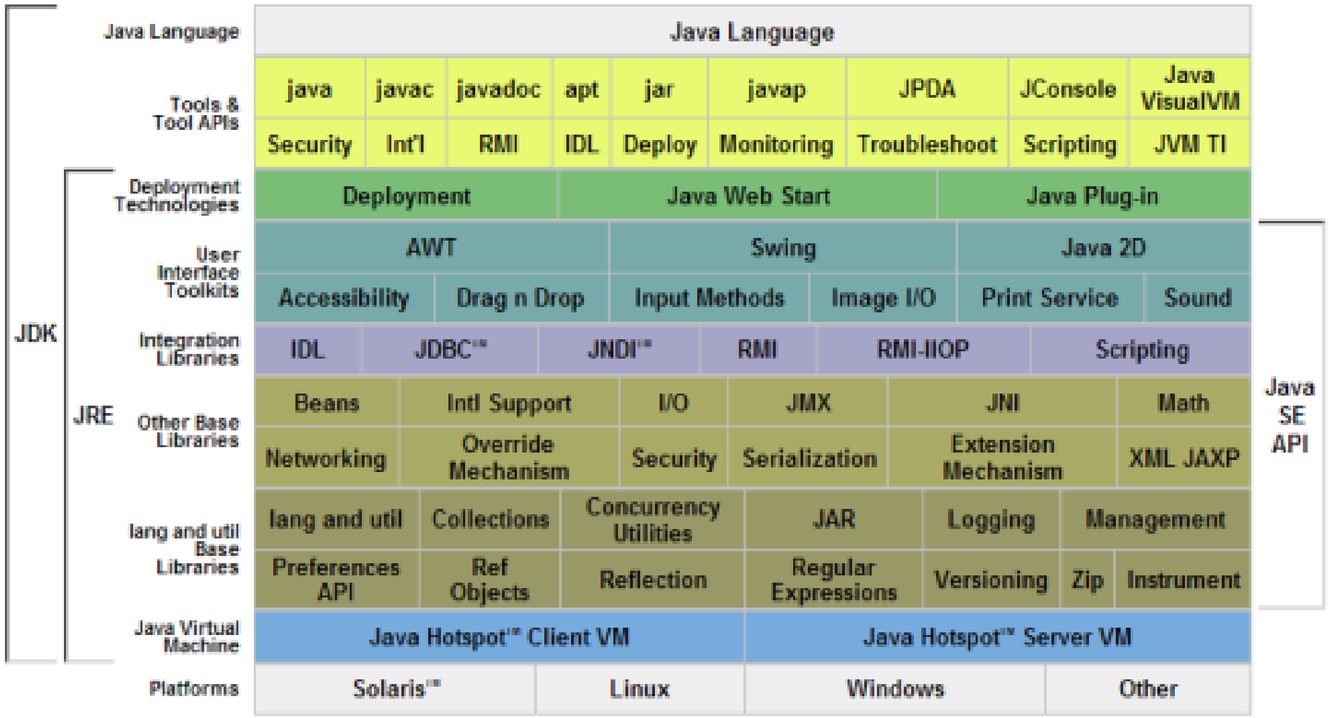
Legend: Old version | Older version, still supported | **Latest version** | Future re

rgupta.mtech@gmail.com

What is Java

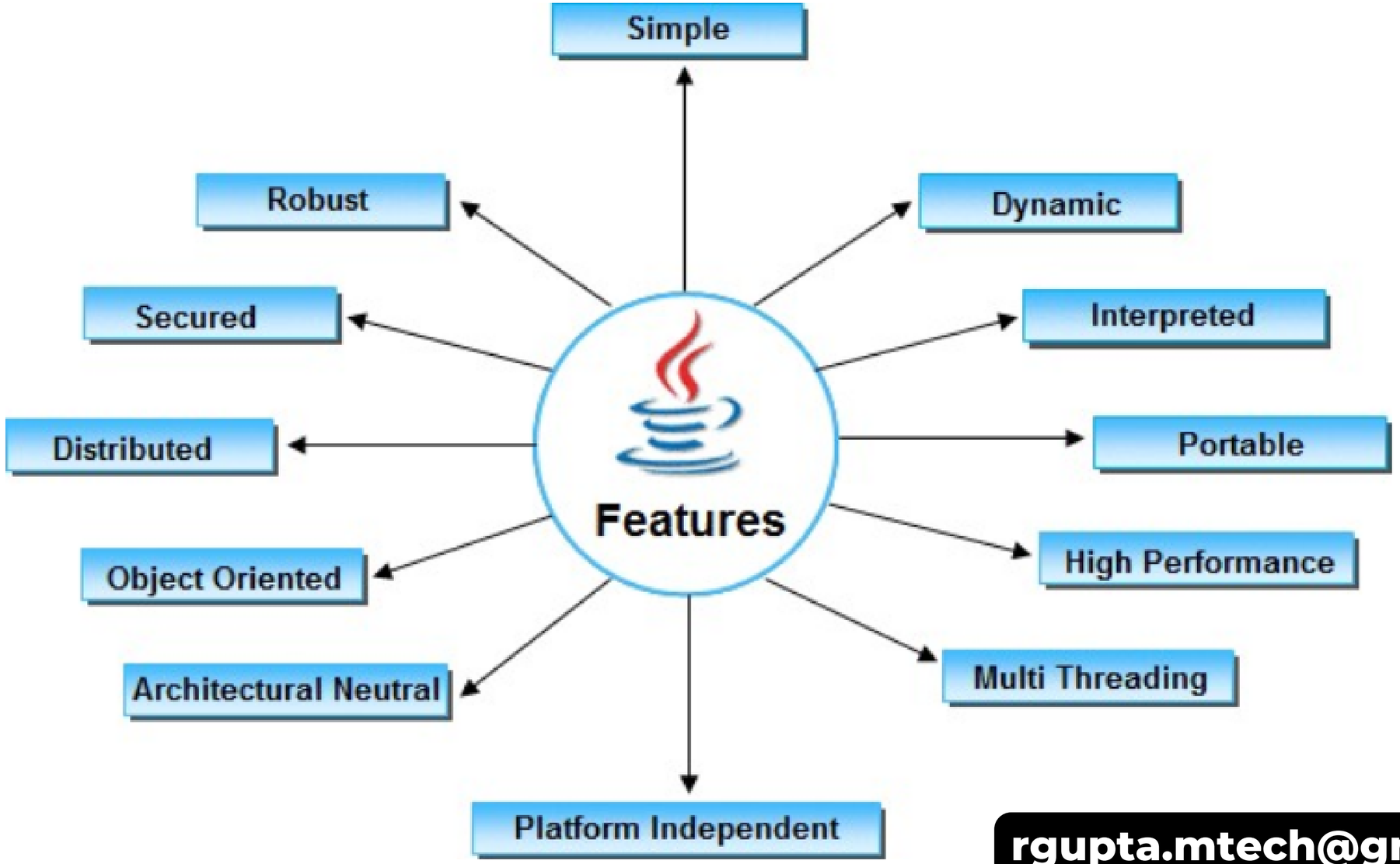
Java=OOPL+JVM+lib

- How Java is different then C++?



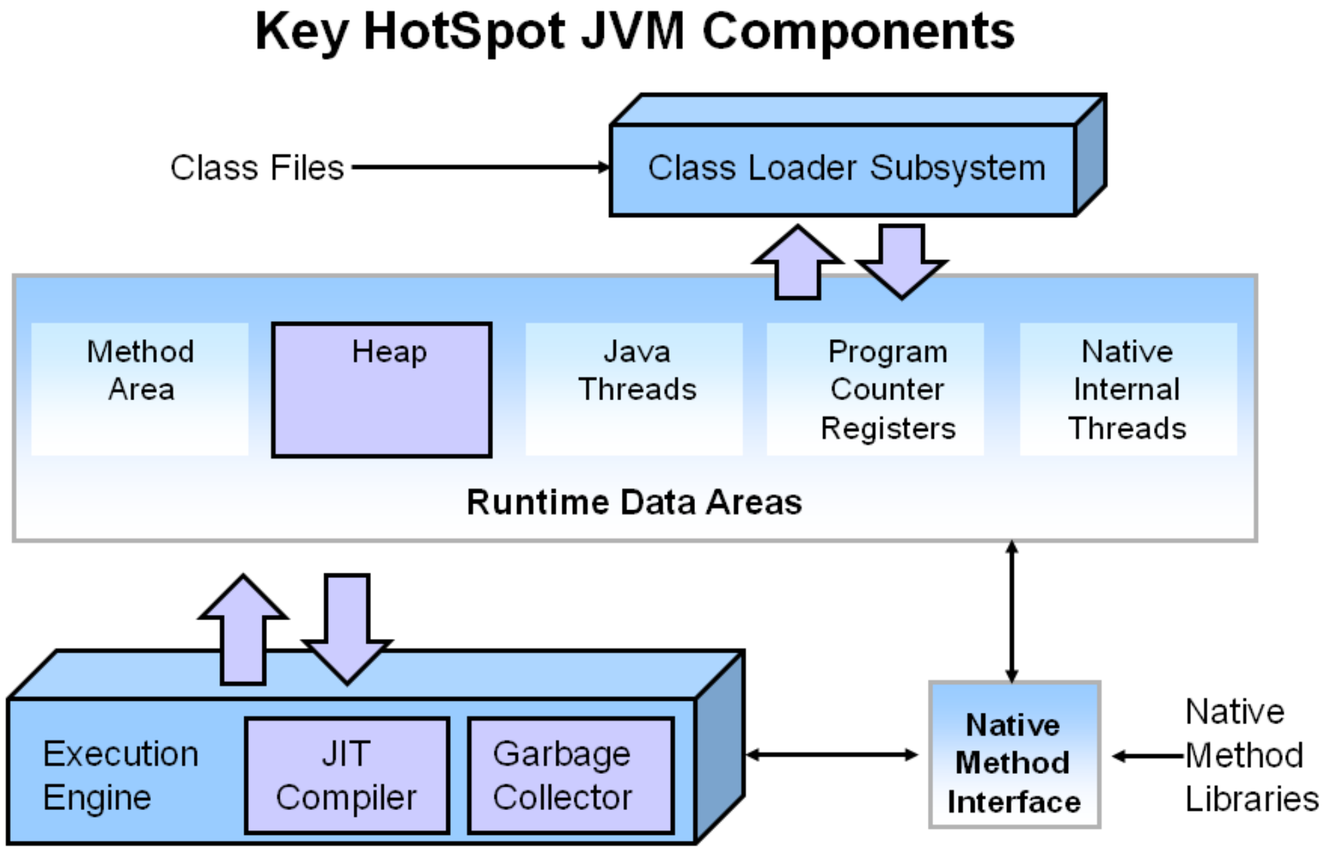
rgupta.mtech@gmail.com

How Sun define Java?



rgupta.mtech@gmail.com

Understanding JVM



rgupta.mtech@gmail.com

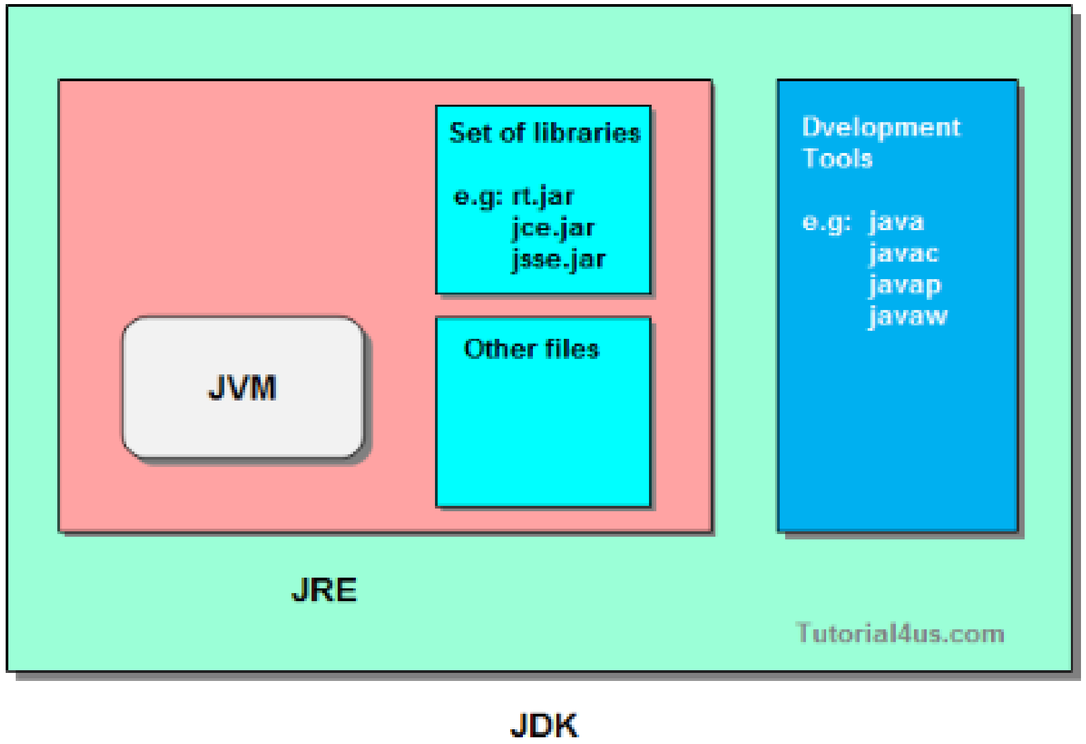
Some common buzzwords

JVM

- **Java Virtual Machine JVM**
- **Pick byte code and execute on client machine Platform dependent…**

JDK

- **Java development kit, developer need it**
- **Byte Code**
- **Highly optimize instruction set, designed to run on JVM Platform independent**
- **Java Runtime environment, Create an instance JVM, must be there on deployment machine.**
- **Platform dependent**



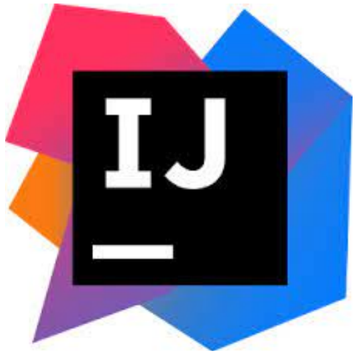
rgupta.mtech@gmail.com

Lab setup and Hello world

Lab set-up

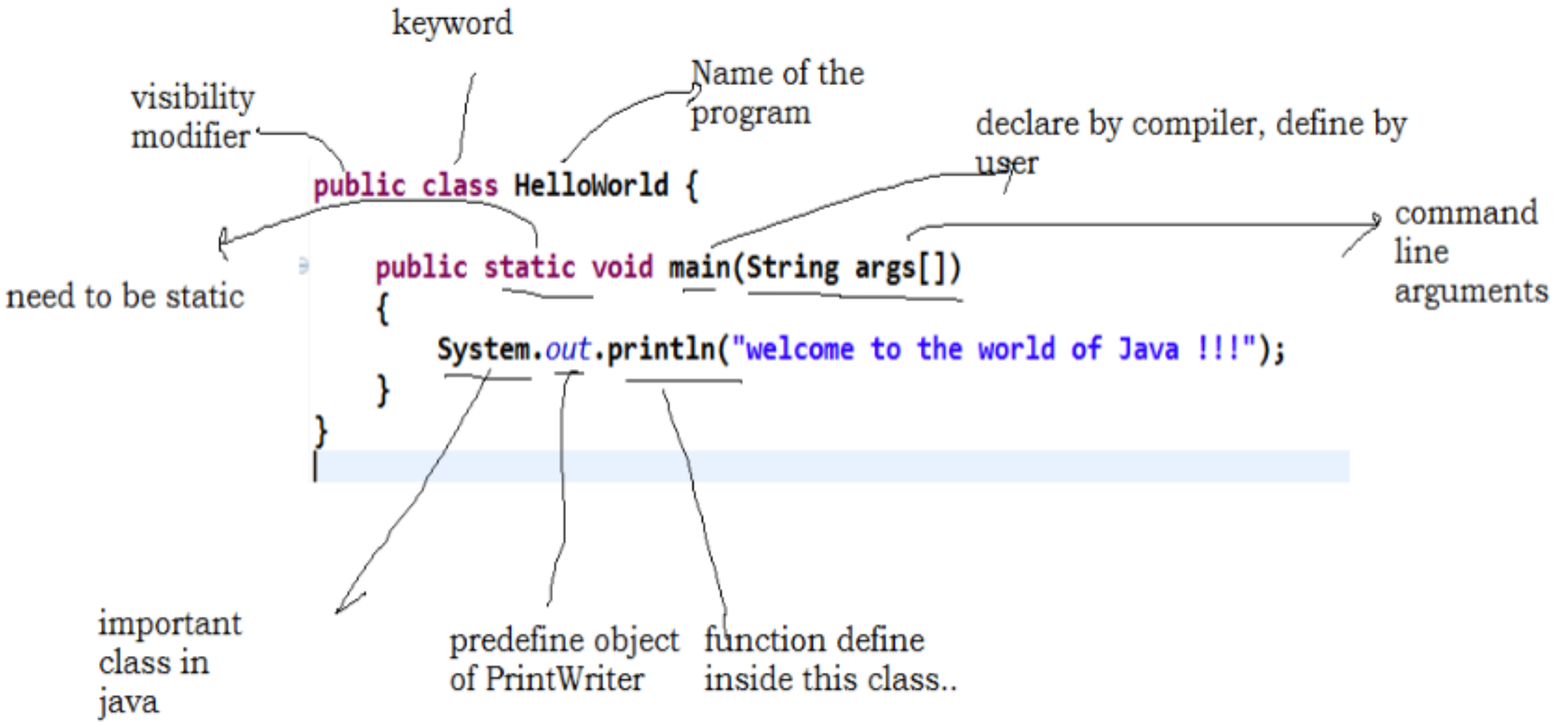
- **Java 17**
- **intellij community edition**

```java
public class HelloWorld {

    public static void main(String args[])
    {
        System.out.println("welcome to the world of Java !!!");
    }
}
```



rgupta.mtech@gmail.com

Analysis of Hello World



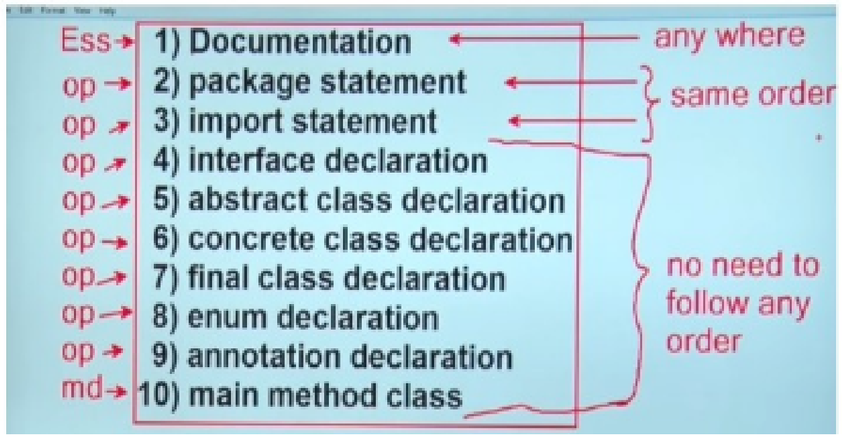
rgupta.mtech@gmail.com

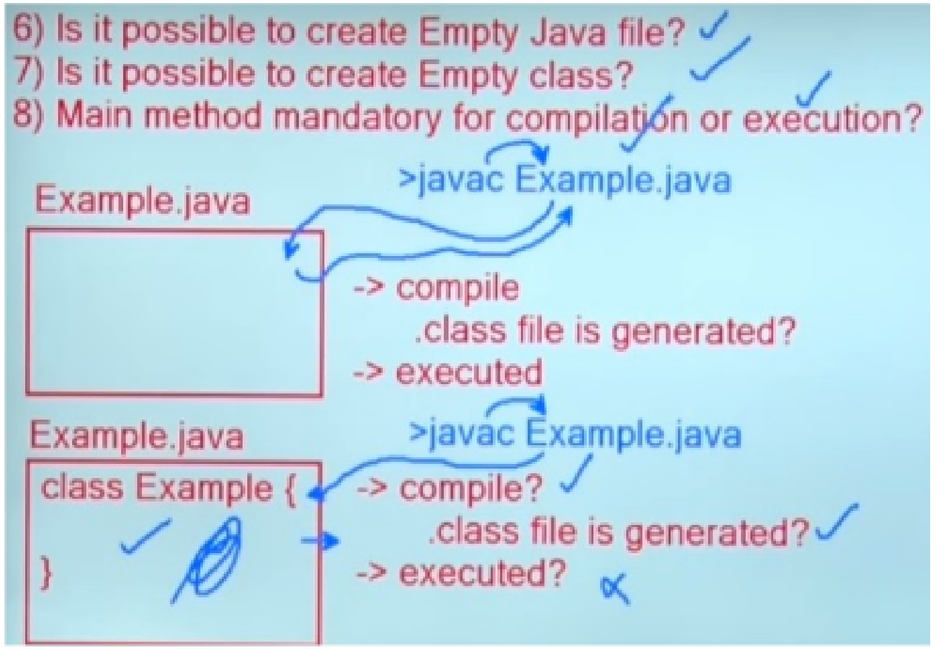
Some basics rules about java classes

1) Java Source file structure

2) Order of placing package and import statement
3) Order of placing interface, class, enum, annotation, doc comment

4) How many package and import statements are allowed in a single java file?
5) Why only one package statement is allowed in and why multiple import statements are allowed?



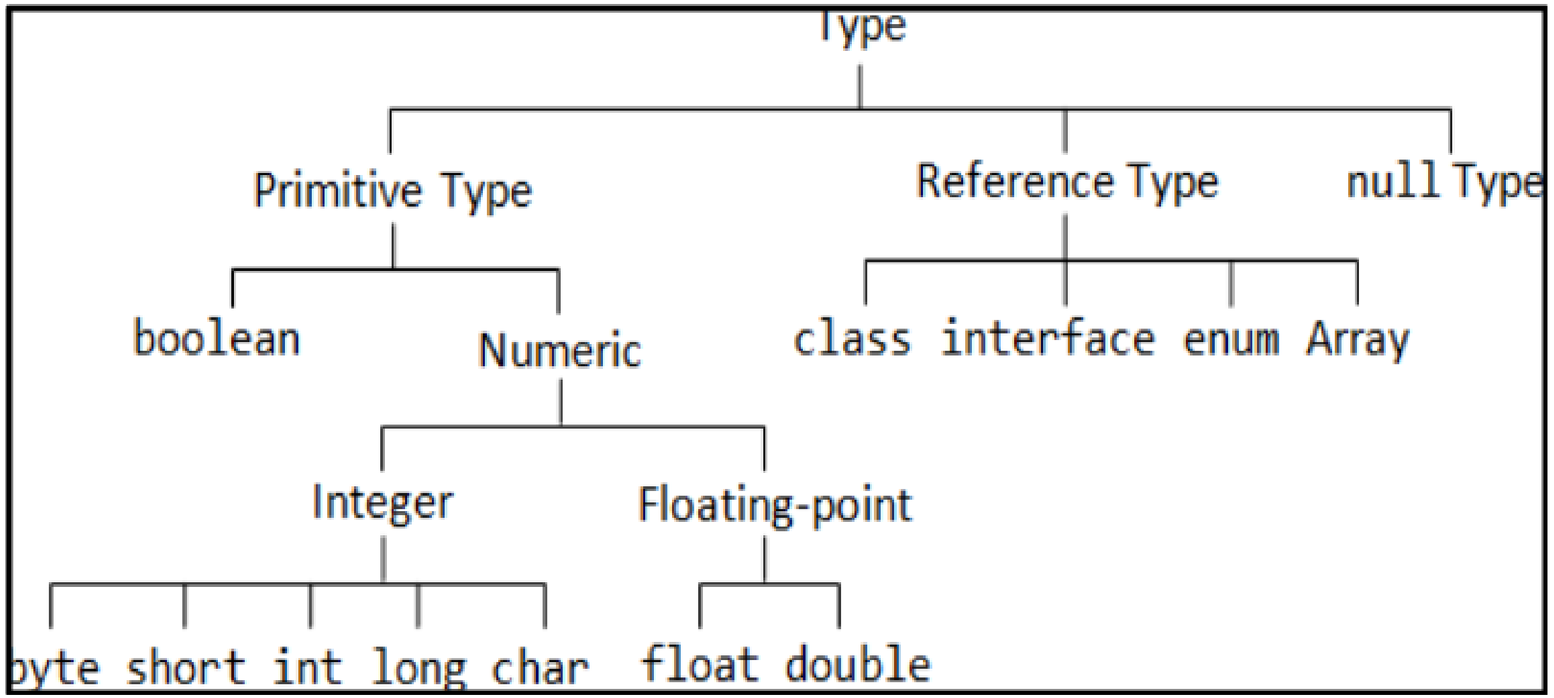


rgupta.mtech@gmail.com

Java Data Type

2 type

- **Primitive data type**
- **Reference data type**



rgupta.mtech@gmail.com

Primitive data type

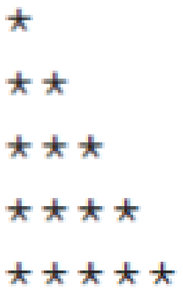
boolean	either true of false
char	16 bit Unicode 1.1
byte	8-bit integer (signed)
short	16-bit integer (signed)
int	32-bit integer (signed)
Long	64-bit integer (singed)
float	32-bit floating point (IEEE 754-1985)
double	64-bit floating point (IEEE 754-1985)

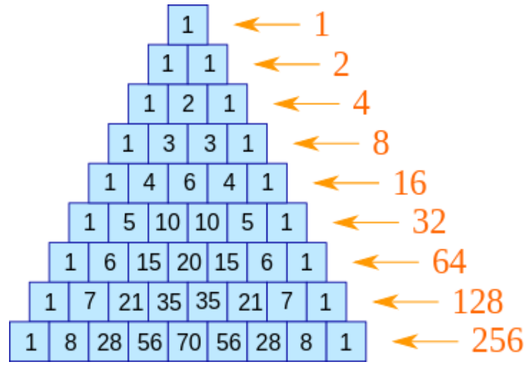
Java uses Unicode to represent characters internally

rgupta.mtech@gmail.com

Procedural Programming

- if..else
- switch
- Looping; for, while, do..while as usual in Java as in C/C++
- Don’t mug the program/logic
- Follow dry run approach
- Try some programms:
- Create
- Factorial program
- Prime No check
- Date calculation





rgupta.mtech@gmail.com

Array are object in java

How Java array different frrom C/C++ array?

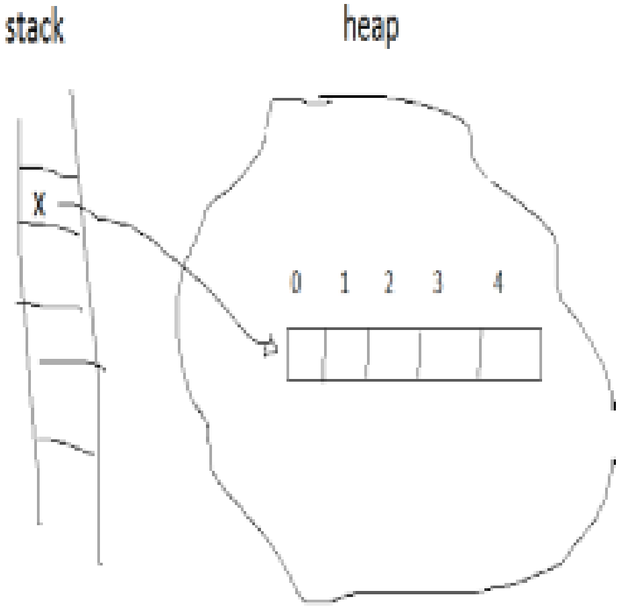
One Dimensional array

Initialization `int a[] = new int [12];`

Value	1	2	3	4	5	6	7	8	9	10	11	12
Index	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a11[11]

```
System.out.print(a[5]);
```

Output: 6



Java Array: its different then C/C++

rgupta.mtech@gmail.com

What is Object Orientation?

- OO is a way of looking at a software system as a collection of interactive objects

Reality

Tom's House

Tom

Tom's Car

Model

rgupta.mtech@gmail.com

What is Object?

Informally, an object represents an entity, either physical, conceptual, or software.

- Physical entity

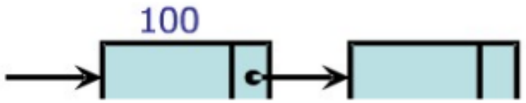
Tom's Car

- Conceptual entity

US$100,000,000,000

Bill Gate's bank account

- Software entity



What is class?

- A class is the blueprint from which individual objects are created.
- An object is an instance of a class.

Object factory

Cookie Cutter

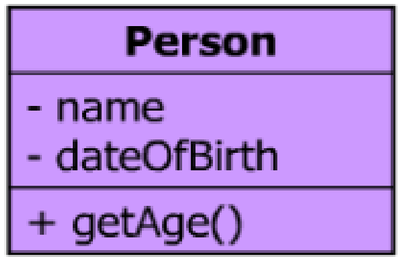
```
public class StudentTest {
  public static void main(String[] args) {
    Student s1 = new Student();
    Student s2 = new Student();
  }
}
```

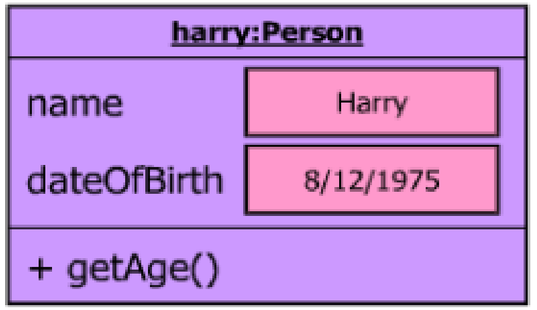
```
- class -

public class Student {
  private String name;
  // …
}
```

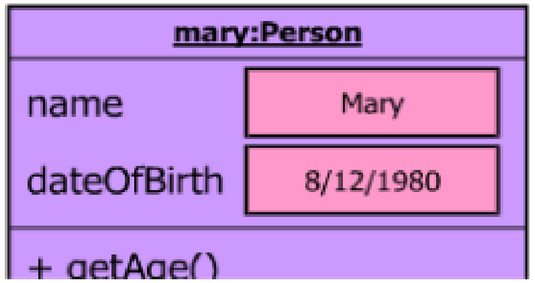
rgupta.mtech@gmail.com

class and objects

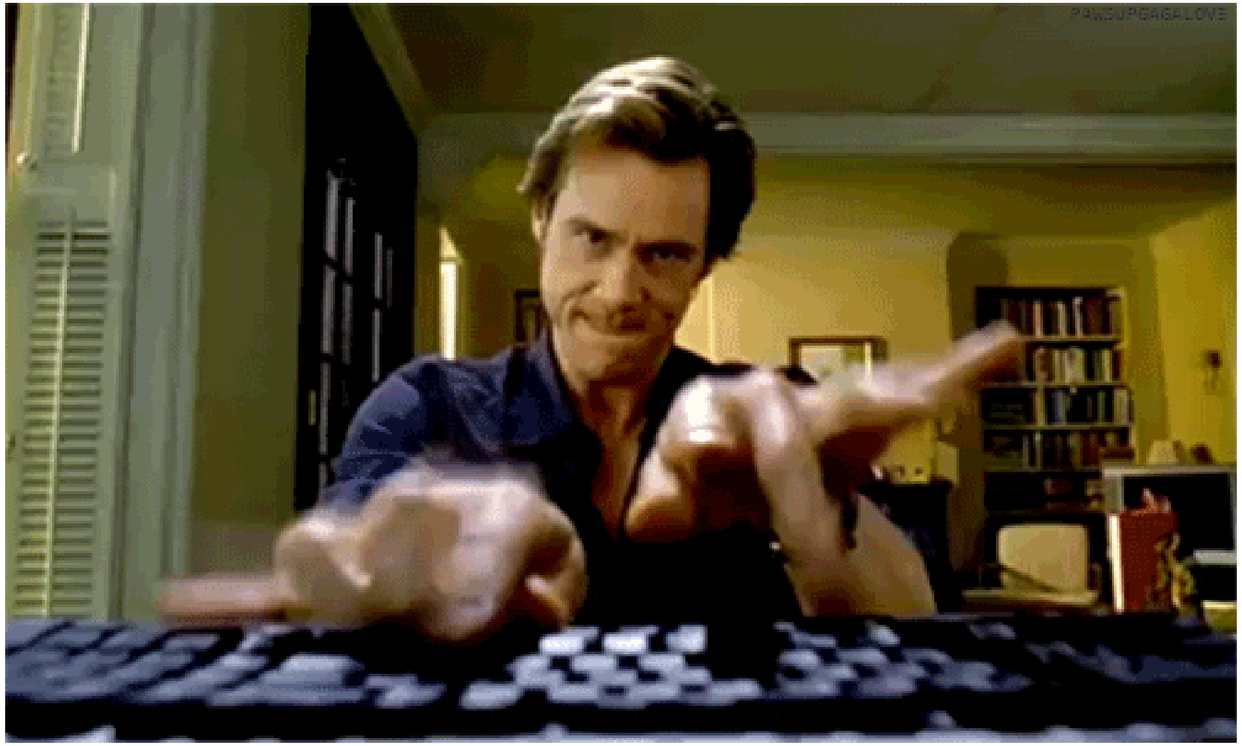
- All the objects share the same attribute names and methods with other objects of the same class
- Each object has its own value for each of the attribute







Programmer are Not typist but thinker



rgupta.mtech@gmail.com

Learning OOPs, SOLID, Design Patterns step by step

Lib & Framework

BTech/MTech

12th class

Design patterns

10th class

SOLID

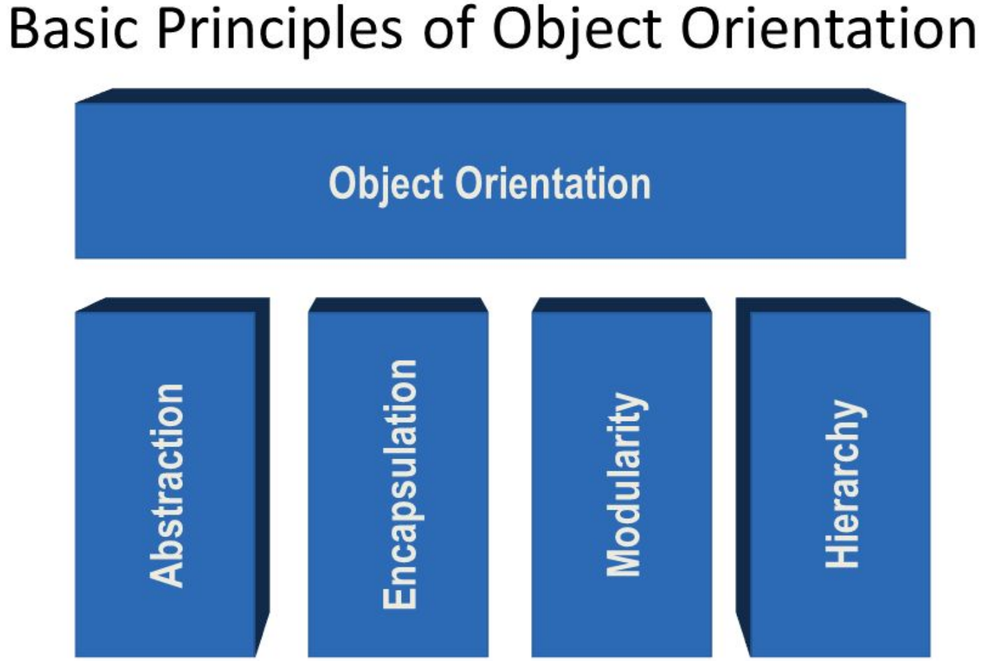
5th class

clear understanding of OOPs concepts

rgupta.mtech@gmail.com

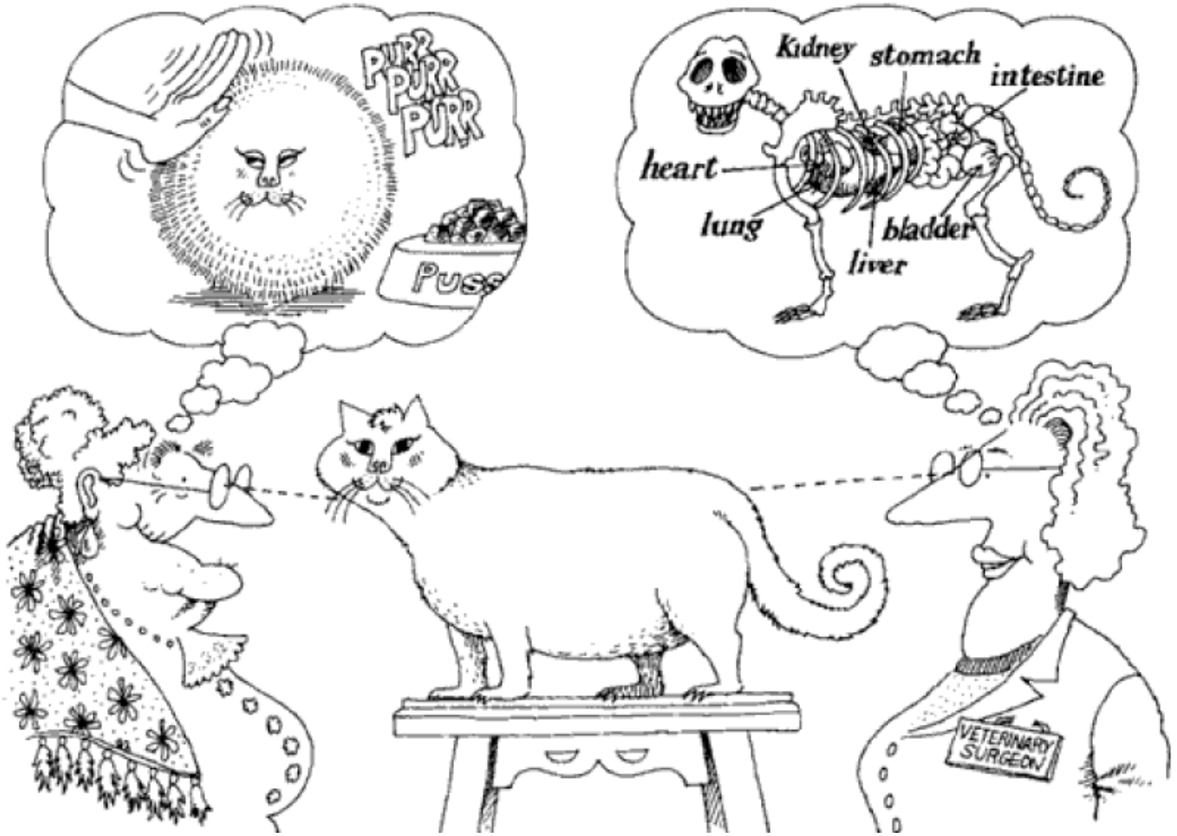
Pillars of object oriented programmin

Abstraction
Encapsulation
Modularity
Hierarchy

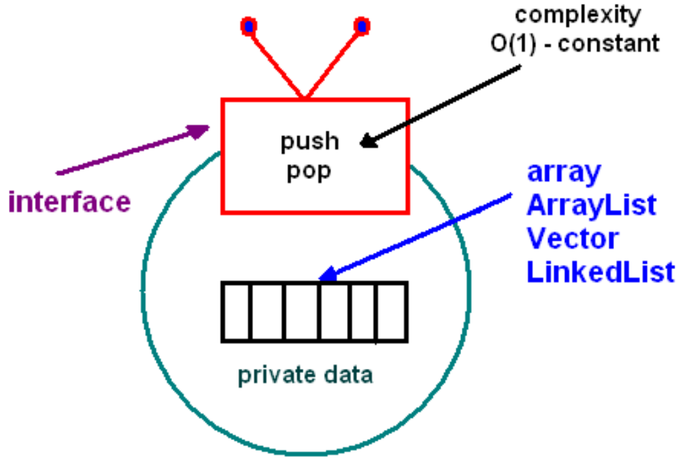


rgupta.mtech@gmail.com

Abstraction



Abtraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer



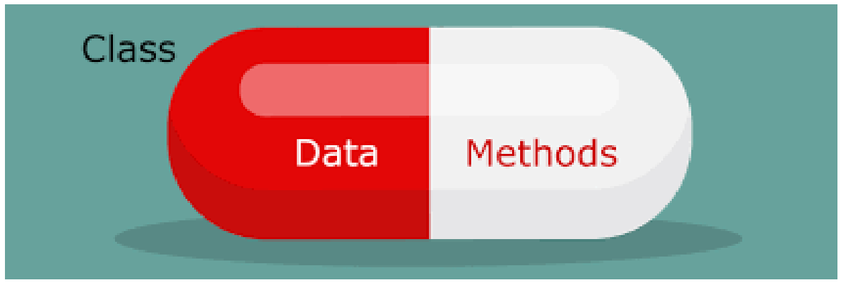
rgupta.mtech@gmail.com

Encapsulation

How to implement Encapsulation provide private visibility to an instance variable

```
class Account{
private int account_number;
private int account_balance;

public void show Data(){
// code to show data
}

public void deposit(int a){
```

Encapsulation the process of restructuring access to inner implementation details of a class. Internal not exposed to the outside world For example, sealed mobile, if you open it guarantee is violated



rgupta.mtech@gmail.com

Abstraction vs Encapsulation

Abstraction is the Design principle of separating interface (not java keyword) from implementation so that the client only concern with the interface

Abstraction in Java programming can be achieved using an interface or abstract class

Abstraction and Encapsulation are complimentary concepts

Encapsulation the process of restructuring access to inner implementation details of a class.

Internal not exposed to the outside world For example, sealed mobile, if you open it guarantee is violated

How to implement Encapsulation provide private visibility to an instance variable

Ability to refactor/change internal code without breaking other client code

Abstraction and Encapsulation are complementary concepts. Through encapsulation only we are able to enclose the components of the object into a single unit and separate the private and public members. It is through abstraction that only the essential behaviors of the objects are made visible to the outside world.

rgupta.mtech@gmail.com

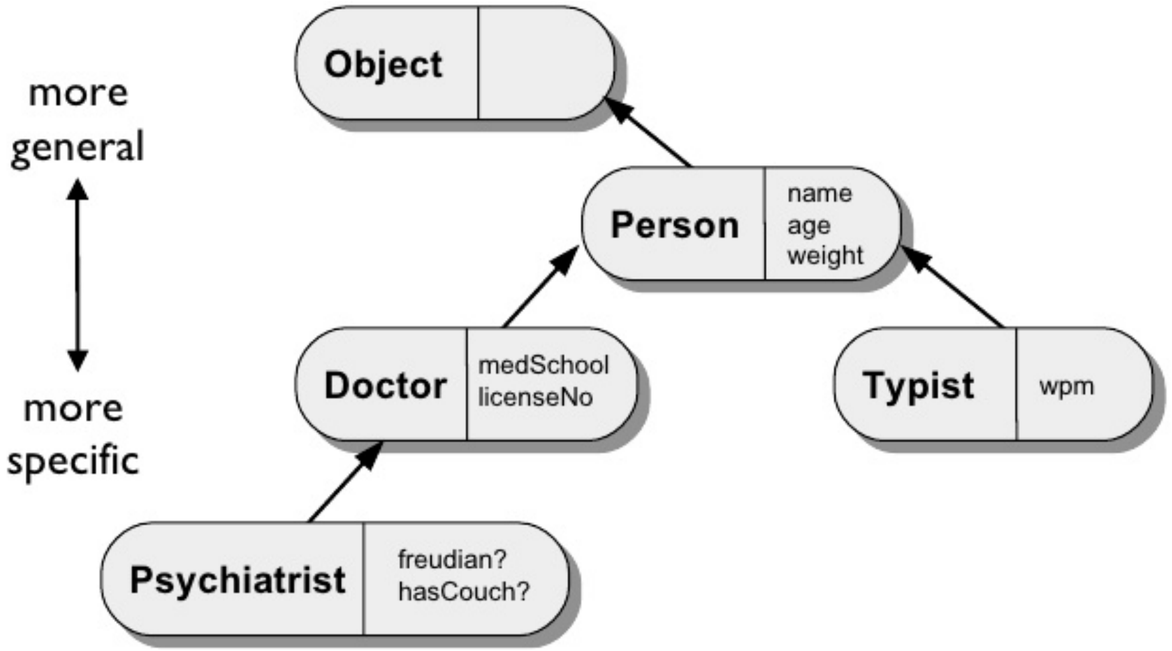
Abstraction vs Encapsulation

Abstraction is a general concept formed by extracting common features from specific examples or The act of withdrawing or removing something **unnecessary**.	Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both **safe from outside interference** and **misuse**.
You can use abstraction using **Interface** and **Abstract** Class	You can implement encapsulation using **Access Modifiers** (Public, Protected & Private)
Abstraction solves the problem in **Design** Level	Encapsulation solves the problem in **Implementation** Level
For simplicity, abstraction means hiding implementation using Abstract class and Interface	For simplicity, encapsulation means hiding data using getters and setters

rgupta.mtech@gmail.com

Hierarchy

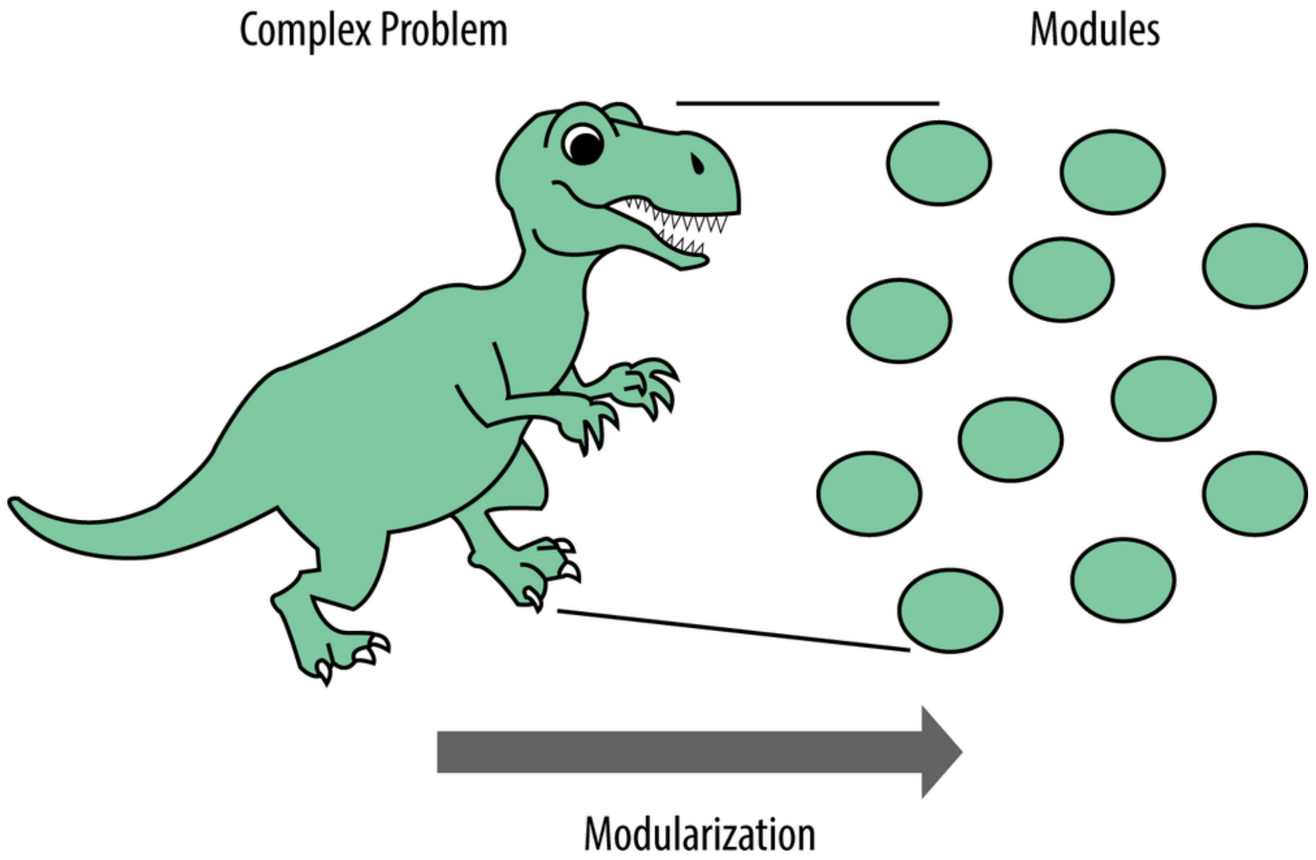
Hierarchy give rise to inheritance and polymorphism



rgupta.mtech@gmail.com

Modularity

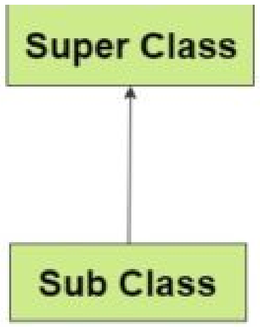
When a class derives from another class. The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods. An inherited class is defined by using the extends keyword

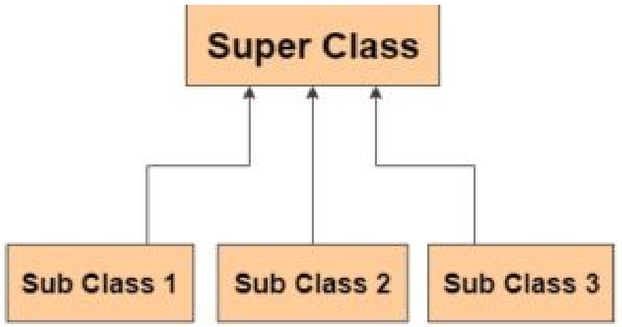


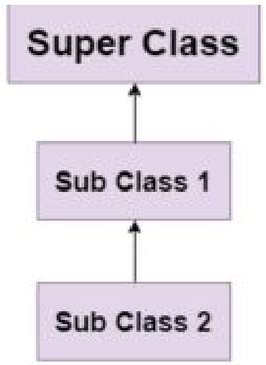
rgupta.mtech@gmail.com

Inheritance

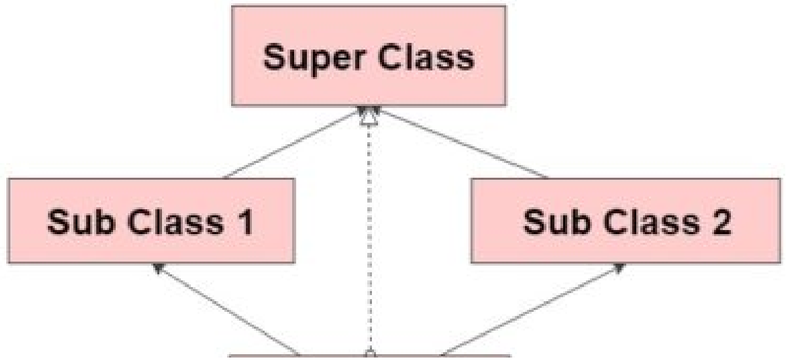
When a class derives from another class. The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods. An inherited class is defined by using the extends keyword



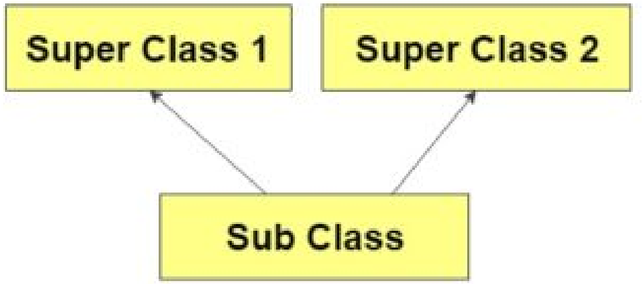




Hybrid Inheritance



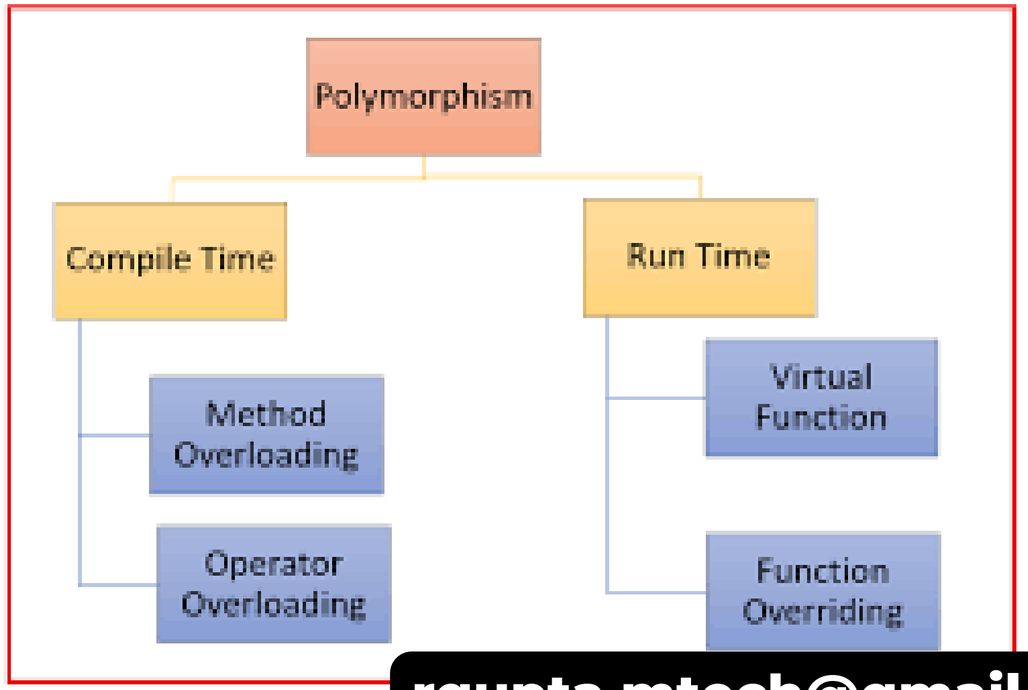
Multiple Inhertance



rgupta.mtech@gmail.com

Polymorphism

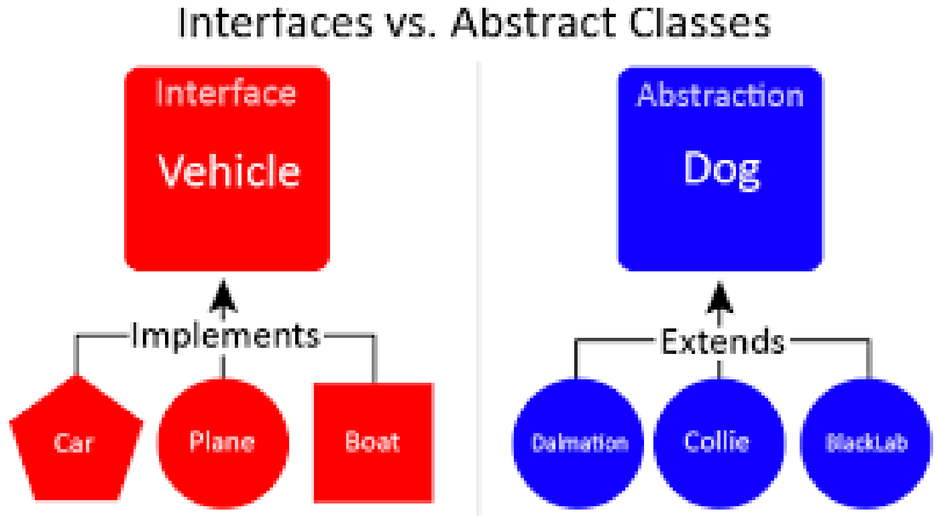
Polymorphism is one of the core concepts of object-oriented programming (OOP) and describes situations in which something occurs in several different forms. In computer science, it describes the concept that you can access objects of different types through the same interface.



rgupta.mtech@gmail.com

Interface vs Abstract class

	Interface	Abstract Class
Constructors	✗	✓
Static Fields	✓	✓
Non-static Fields	✗	✓
Final Fields	✓	✓
Non-final Fields	✗	✓
Private Fields & Methods	✗	✓
Protected Fields & Methods	✗	✓
Public Fields & Methods	✓	✓
Abstract methods	✓	✓
Static Methods	✓	✓
Final Methods	✗	✓
Non-final Methods	✓	✓
Default Methods	✓	✗



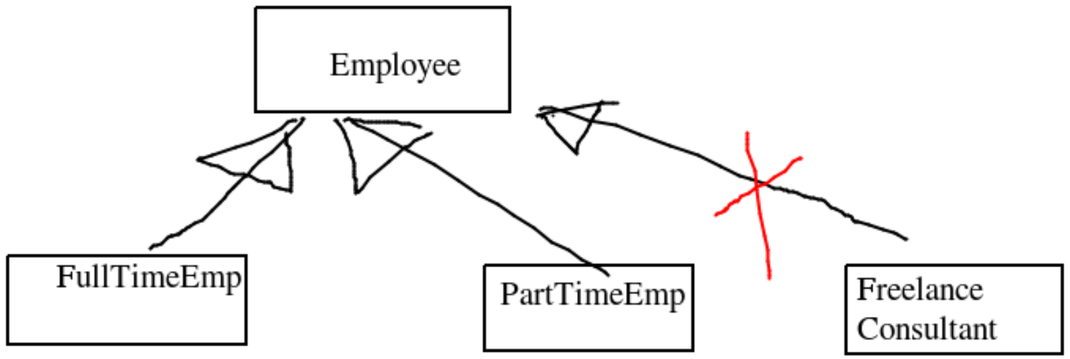
rgupta.mtech@gmail.com

Interface vs Abstract class

Interface: Specification of a behavior The interface represents some features of a class

What an object can do?

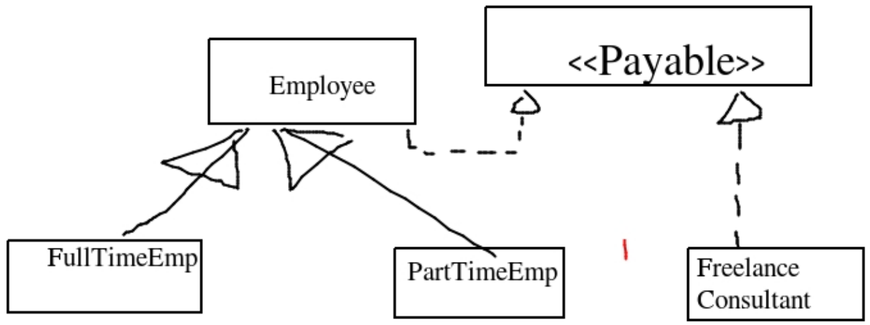
Standardization of a behavior



Abstract class: generalization of a behavior The incomplete class that required further specialization

What an object is?

IS-A SavingAccount IS-A Account



When we should go for interface and when we should go for abstract class?
Interface break the hierarchy

rgupta.mtech@gmail.com

Session 2:
Basic OOPs, Inheritance, Overriding, Overloading, Polymorphism

rgupta.mtech@gmail.com

What can goes inside an class?

```
public class A {

    int i;          //  instance variable

    static int j;   // static variable

    //method in class
    public void foo(){
        int i;      //local variable
    }

    public A(){}            //default constructor

    public A(int j)         //parameterized ctr
    {
        //........
    }

    //getter and setter
    public int getI(){return i;}
    public void setI(int i){this.i=i;}
}
```

rgupta.mtech@gmail.com

Creating Classes and object

```
class Account{

    public int id;
    public double balance;

    //.......
    //......

}

public class AccountDemo{
    public static void main(String[] args) {
        Account ac=new Account();
        ac.id=22;
    }
}
```

killing encapsulation

rgupta.mtech@gmail.com

Correct way?

```
class Account{
    private int id;
    private double balance;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }
}

public class AccountDemo{
    public static void main(String[] args) {
        Account ac=new Account();
        //ac.id=22; will not work
        ac.setBalance(2000);//correct way
    }
}
```

rgupta.mtech@gmail.com

Java Constructor

- Initialize state of the object
- Special method have same name as that of class
- Can’t return anything
- Can only be called once for an object
- Can be private
- Can’t be static*
- Can overloaded but cant overridden*
- Three type of constructors
  - Default,
  - Parameterized and
  - Copy constructor

```
class Account{
    private int id;
    private double balance;

    //default ctr
    public Account() {
        //........
    }

    //parameterized ctr
    public Account(int i, double b) {
        this.id=i;
        this.balance=b;
    }

    //copy ctr
    public Account(Account ac) {
            //........
        }
}
```

rgupta.mtech@gmail.com

Need of “this” ?

```
class Account{
   private int id;
   private double balance;

   //default ctr
   public Account() {
       //........
   }

   //parameterized ctr
   public Account(int id, double balance) {
       id=id;
       balance=balance;
   }

   //copy ctr
   public Account(Account ac) {
           //........
       }
```

which id assigned to which id ?

- **Which id assigned to which id?**
- **“this” is an reference to the current object required to differentiate local variables with instance variables**

rgupta.mtech@gmail.com

“this” used to resolve confusion…

```
class Account{

    private int id;
    private double balance;

    //default ctr
    public Account() {
        //........
    }

    //parameterized ctr
    public Account(int id, double balance) {
        this.id=id;
        this.balance=balance;
    }

    //copy ctr
    public Account(Account ac) {
```

refer to instance variable

rgupta.mtech@gmail.com

this : Constructor chaining?

```
class Account{

    private int id;
    private double balance;

    //default ctr
    public Account() {
        this(22,555.0);
    }

    //parameterized ctr
    public Account(int id, double balance) {
        this.id=id;
        this.balance=balance;
    }

    //copy ctr
    public Account(Account ac) {
            //
```

Calling one constructor from another ?

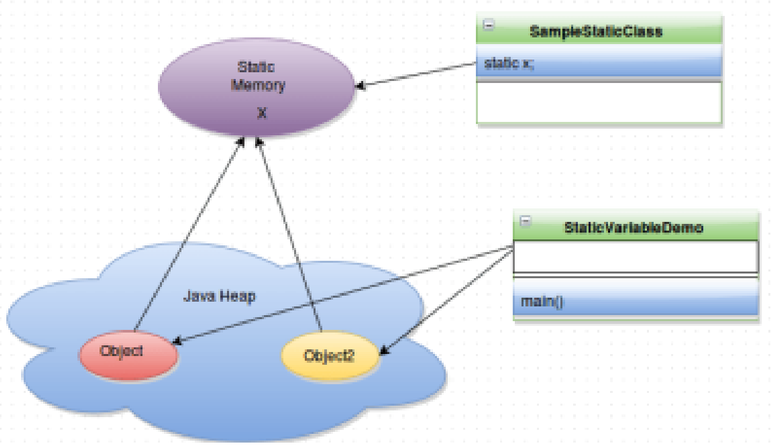
rgupta.mtech@gmail.com

static method/variables

- Instance variable –per object while static variable are per class
- Initialize and define before any objects
- Most suitable for counter for object
- Static method can only access static data of the class
- For calling static method we don't need an object of that class

Now guess why main was static?

How to count number of account object in the memory?



rgupta.mtech@gmail.com

static method/variables

```
class Account{

    private int id;
    private double balance;

    // will count no of account in application
    private static int totalAccountCounter=0;

    public Account(){
        totalAccountCounter++;
    }

    public  static int  getTotalAccountCounter(){
        return totalAccountCounter;
    }

}
```

static variable

static method

We can not access instance variable in static method but can access static variable in instance method

```
Account ac1=new Account();
Account ac2=new Account();

//How maany account are there in application ?

System.out.println(Account.getTotalAccountCounter());

System.out.println(ac1.getTotalAccountCounter());
```

rgupta.mtech@gmail.com

Initialization block

- We can put repeated constructor code in an Initialization block...

- Static Initialization block runs before any constructor and runs only once...

```
class Account{
   private int id;
   private double balance;
   private int accountCounter=0;

   static{
      System.out.println("static block: runs only once ...");
   }

   {
      System.out.println("Init block 1: this runs before any constructor  ...");
   }

   {
      System.out.println("Init block 2: this runs after inti  block 1 , before any const execute  ...");
   }
}
```

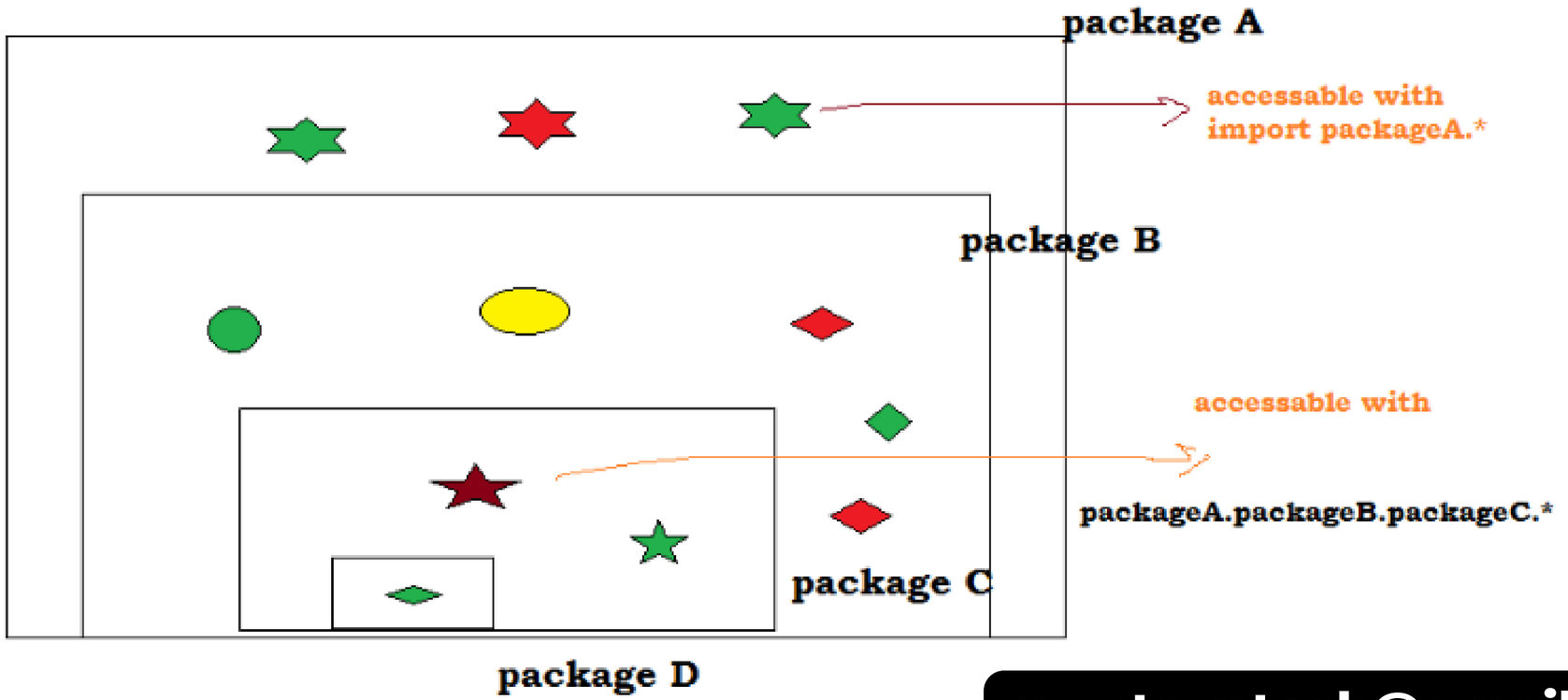
```
class Account{

   private int id;
   private double balance;

   public Account(){
      //this is common code
   }
   public Account(int id , double balance){
      //this is common code

      this.id=id;
      this.balance=balance;
   }

}
```

code repetition.

rgupta.mtech@gmail.com

concept of packages

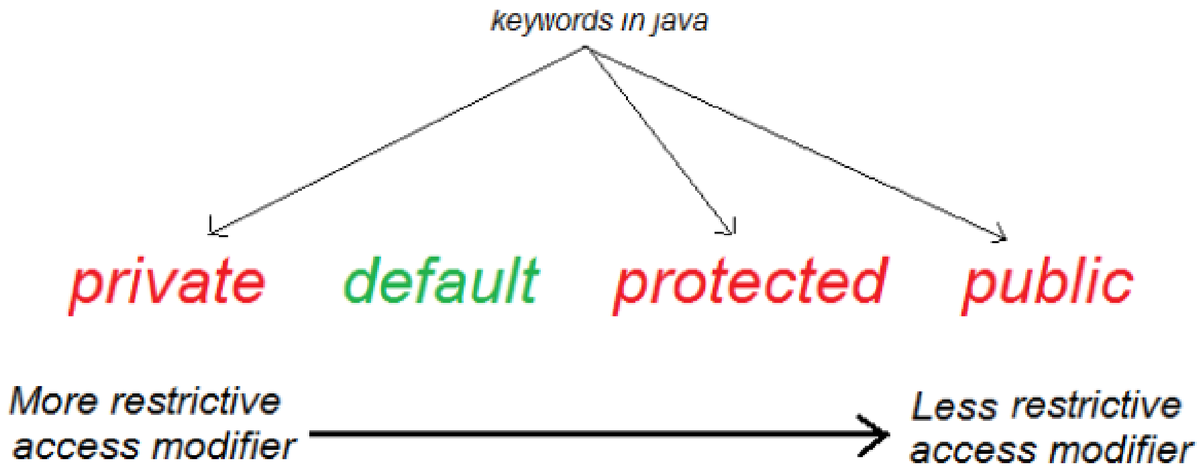
- Packages are Java's way of grouping a number of related classes and/or interfaces together into a single unit.
- Packages act as "containers" for classes.



rgupta.mtech@gmail.com

visibility modifiers

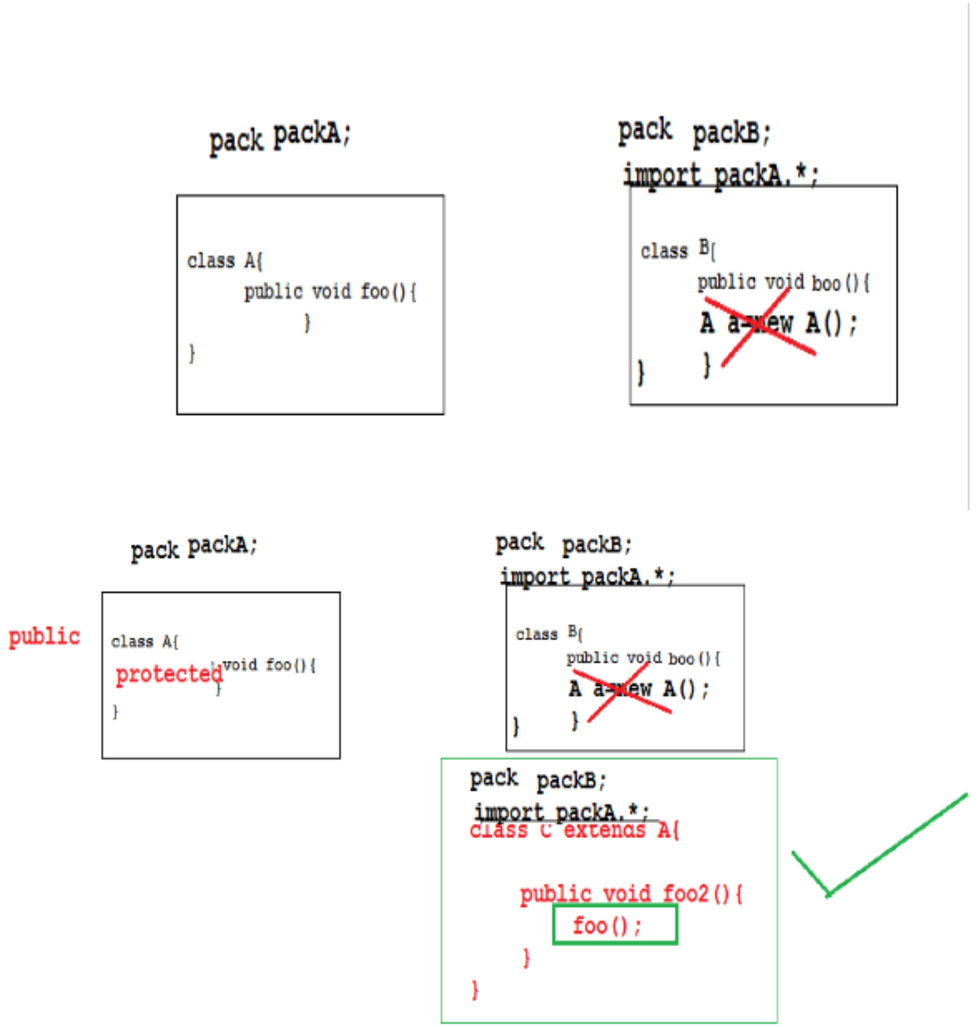
- For instance variable and methods
  - Public
  - Protected
  - Default (package level)
  - Private
- For classes
  - Public and default



rgupta.mtech@gmail.com

visibility modifiers

- class A has default visibility
  - hence can access in the same package only.
- Make class A public, then access it.
- Protected data can access in the same package and all the subclasses in other packages provide class itsef is public



rgupta.mtech@gmail.com

Want to accept parameter from user?

java.util.Scanner (Java 1.5)

```
Scanner stdin = Scanner.create(System.in);  int n = stdin.nextInt();
String s = stdin.next();
```

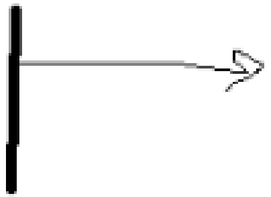
```
boolean b = stdin.hasNextInt()
```

rgupta.mtech@gmail.com

Call by value

- **Java support call by value**
- **The value changes in function is not going to reflected in the main.**

```
public class CallByValue {
    public static void main(String[] args) {
        int i=22;
        int j=33;
        System.out.println("value of i before swapping:"+i);
        System.out.println("value of j before swapping:"+j);
        swap(i,j);
    }

     static void swap(int i, int j) {
        int temp;
        temp=i;
        i=j;
        j=temp;
    }
}
```



not going to change value in main...

rgupta.mtech@gmail.com

Call by reference

- **Java don’t support call by reference.**
- **When you pass an object in an method copy of reference is passed so that we can mutate the state of the object but can’t delete original object itself**

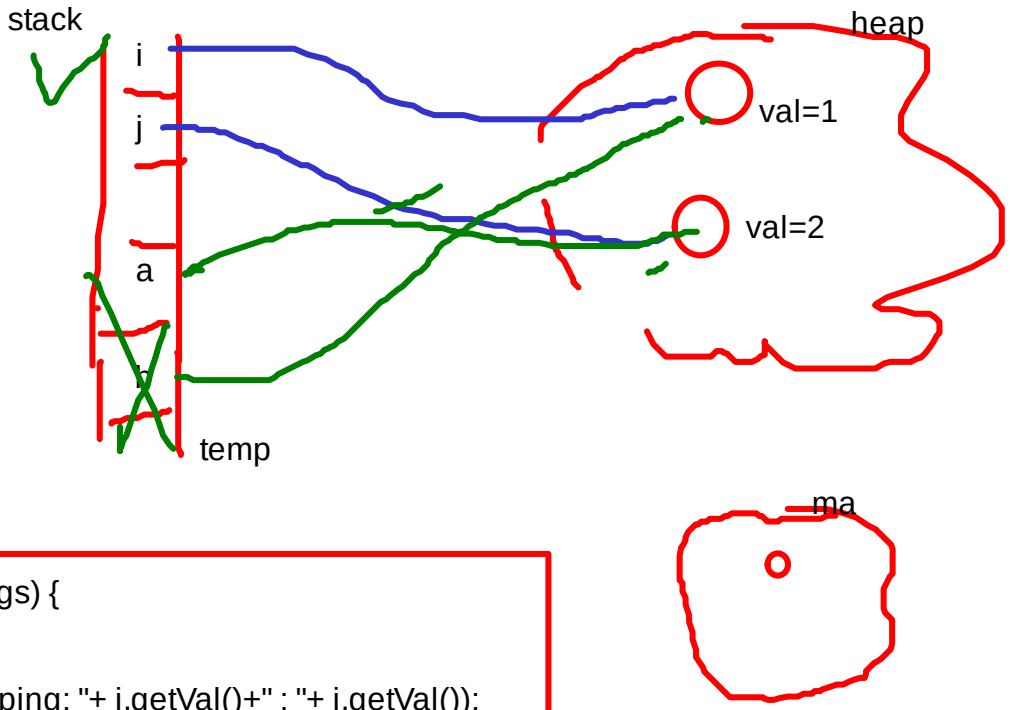
```
class Foo{
    private int i;
    public Foo(int i){
        this.i=i;
    }
    public int getI(){return i;}
    public void setI(int t){i=t;}
}
public class CallByref {

    public static void main(String[] args) {
        Foo f1=new Foo(22);
        Foo f2=new Foo(33);
        swap(f1,f2);
    }

     static void swap(Foo f1, Foo f2) {
        Foo temp;
        temp=f1;
        f1=f2;
        f2=temp;
        // f1.setI(55);
    }
```

do not effect f1 , f2 in main

can change state of f1

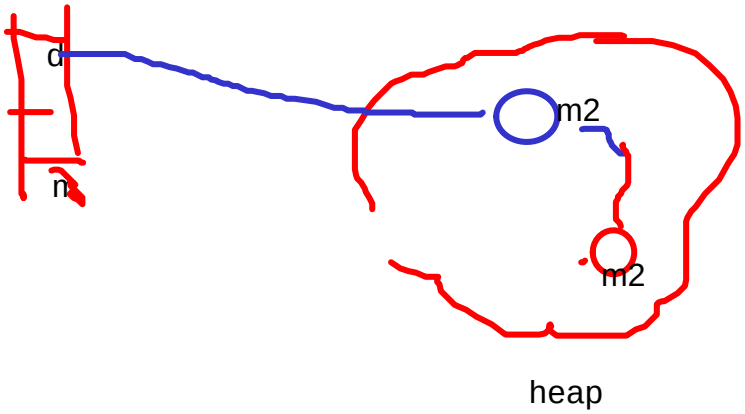
rgupta.mtech@gmail.com

```
class Value{
    private int val;

    public int getVal() {
        return val;
    }
    public void setVal(int val) {
        this.val = val;
    }

    public Value(int val) {
        this.val = val;
    }

}
public class B_CallByRef {
    public static void main(String[] args) {
        Value i=new Value(1);
        Value j=new Value(2);
        System.out.println("before swaping: "+ i.getVal()+" : "+ j.getVal());
        swap(i,j);
        System.out.println("after swaping: "+ i.getVal()+" : "+ j.getVal());

    }
    public static void swap(Value a, Value b){
        Value temp=a;
        a=b;
        b=temp;
    }
}
```



```
class M{
    void foo(){

    }
}
public class Demo2 {
    M m2=new M();

    void foo(){
        m2=null;
        M m=new M();

    }
    void foo2(){
        System.out.println(i);
        m2.foo();
    }
    public static void main(String[] args) {
        Demo2 d=new Demo2();
        d.foo();
        d.foo2();
    }
}
```

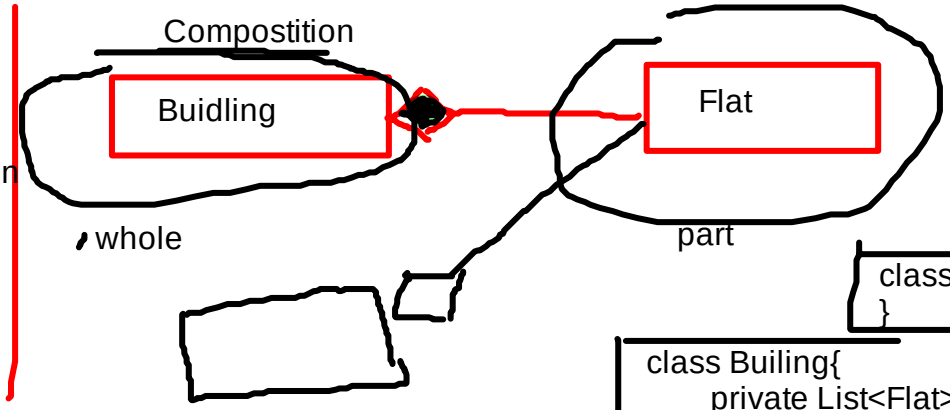
```
class M{
   void foo(){}
}
public class Demo2 {
   M m2=null;
   void foo(){
    //  m2=null;
      M m=new M();
      m2=m;
   }
   void foo2(){
     // System.out.println(i);
      m2.foo();
   }
   public static void main(String[] args) {
      Demo2 d=new Demo2();
      d.foo();
      d.foo2();
   }
}
```



has a(Association)

Compostition

aggregation

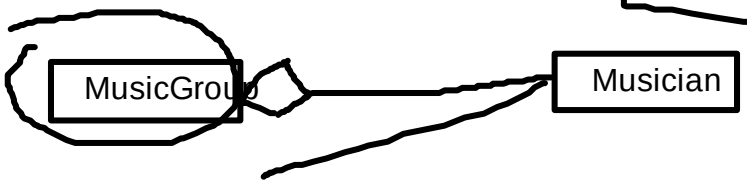


*

```
class LinkedList{

      class Node{

      }

}
```

```
class Flat{
}
class Builing{
      private List<Flat> flats=
            new LinkedList<>();

}
```

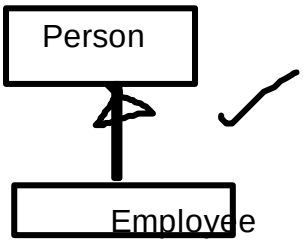
aggregation



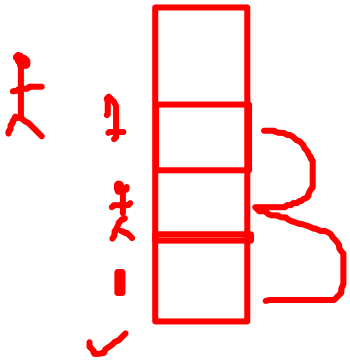
Inheritance

A employee is a person

Emp class is extending person



```
class A{
   int i=6;
   void foo(){
      System.out.println("foo method : "+i);
   }
}
class B extends A{
   int i=66;

   void foo(){
     super.foo();//inhertance support resuablity
      System.out.println("foo2 of class B "+j +" : "+i);
   }
}
public class B_InheritanceEx {
   public static void main(String[] args) {
      A a=new B();
      a.foo();
   }
}
```



```
class A{
   int i=6;
   public A(int i){
      this.i=i;
   }
}
class B extends A{
   int j=66;
   public B(int i, int j){
      this.j=j;
   }
}
```

```
class A{
   int i=6;
   public A(int i){
      this.i=i;
   }
}
class B extends A{
   int j=66;
   public B(int i, int j){
      super(i);
      this.j=j;
   }
}

public class B_InheritanceEx {
   public static void main(String[] args) {
      A a=new B(3,7);
   }
}
```

```
class A{
    int i=6;
    public A(int i){
        this.i=i;
    }
}
class B extends A{
    int j=66;
    public B(int i, int j){
        this.j=j;
    }
}
```

```
class A{
    int i=6;
    public A(int i){
        this.i=i;
    }
}
class B extends A{
    int j=66;
    public B(int i, int j){
        super()
        this.j=j;
    }
}
```

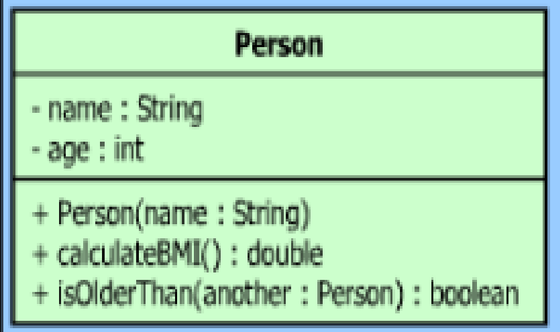
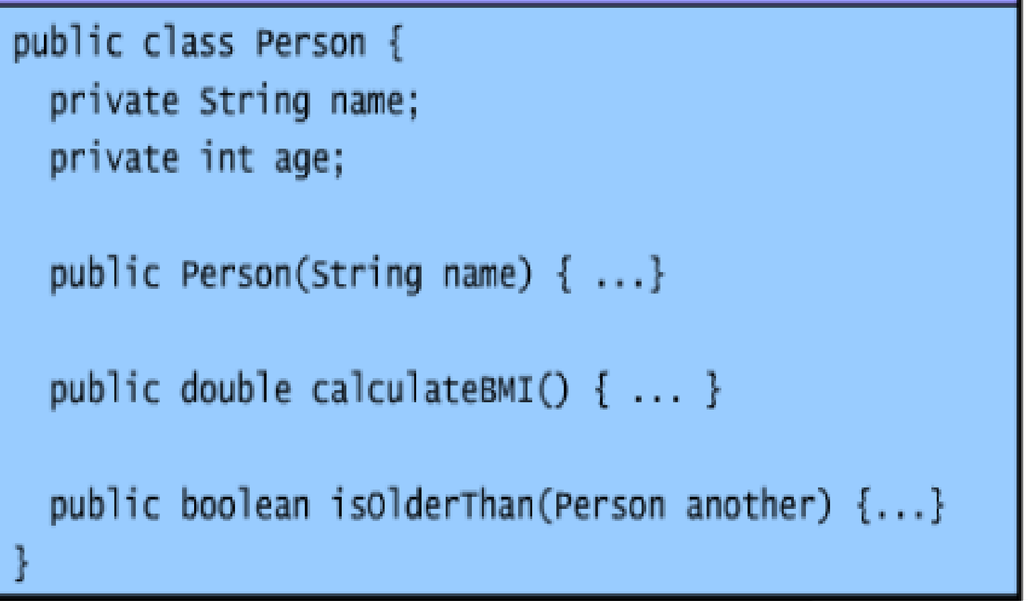
```
class A{
    A(int a){}
}

class B extends A{

    public B(){
        super();

    }

}
```

Session 3
Advance Object Orientation

rgupta.mtech@gmail.com

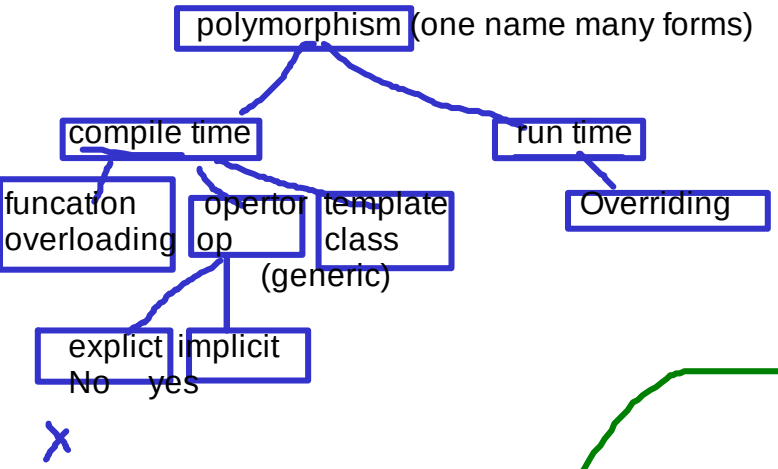
UML diagram Basics

- **UML 2.0 aka modeling language has 12 type of diagrams**
- **Most important once are class diagram, use case diagram and sequence diagram.**
- **You should know how to read it correctly**

UML	Implementation
	


```
public class Person {
  private String name;
  private int age;

  public Person(String name) { ...}

  public double calculateBMI() { ... }

  public boolean isOlderThan(Person another) {...}
}
```

rgupta.mtech@gmail.com



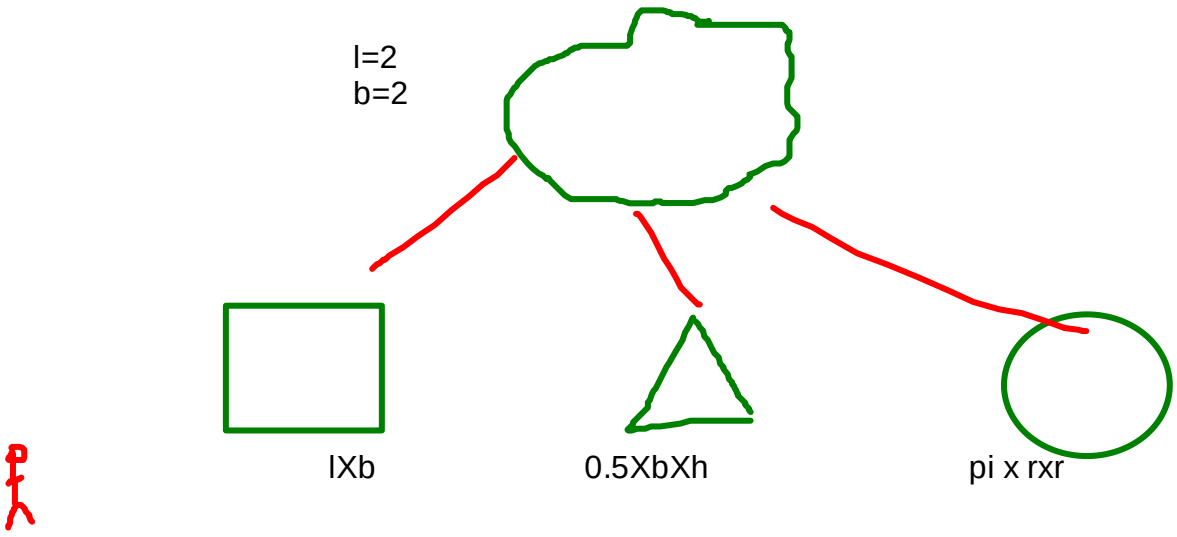
Calcuator

add(int a, int b)
add(int a, int b, int b)

Run time polymophism...

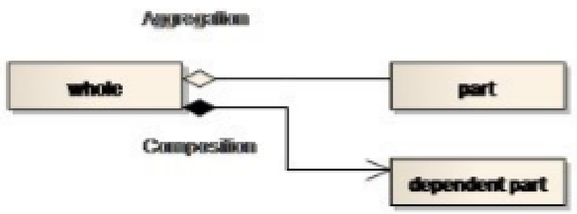
```
class  Calculator{
   long add(int a, long b){
      return a+b;
   }
   long add(long a, int b){
      return a+b;
   }
}
public class A_ExOverloading {
   public static void main(String[] args) {
      Calculator c=new Calculator();
      long result = c.add(3,2);
      System.out.println(result);
   }
}
```

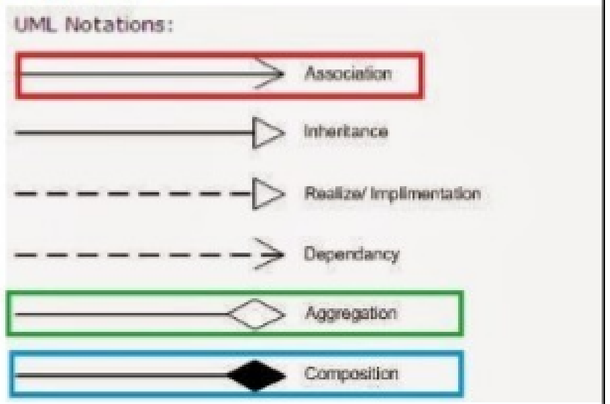
c.add(2,3L)



Relationship between Objects

- USE-A
  - Passanger using metro to reach from office from home
- HAS-A (Association)
  - Compostion
    - Flat is part of Durga apartment
  - Aggregation
    - Ram is musician with RockStart musics group
- IS-A
  - Empoloyee is a person

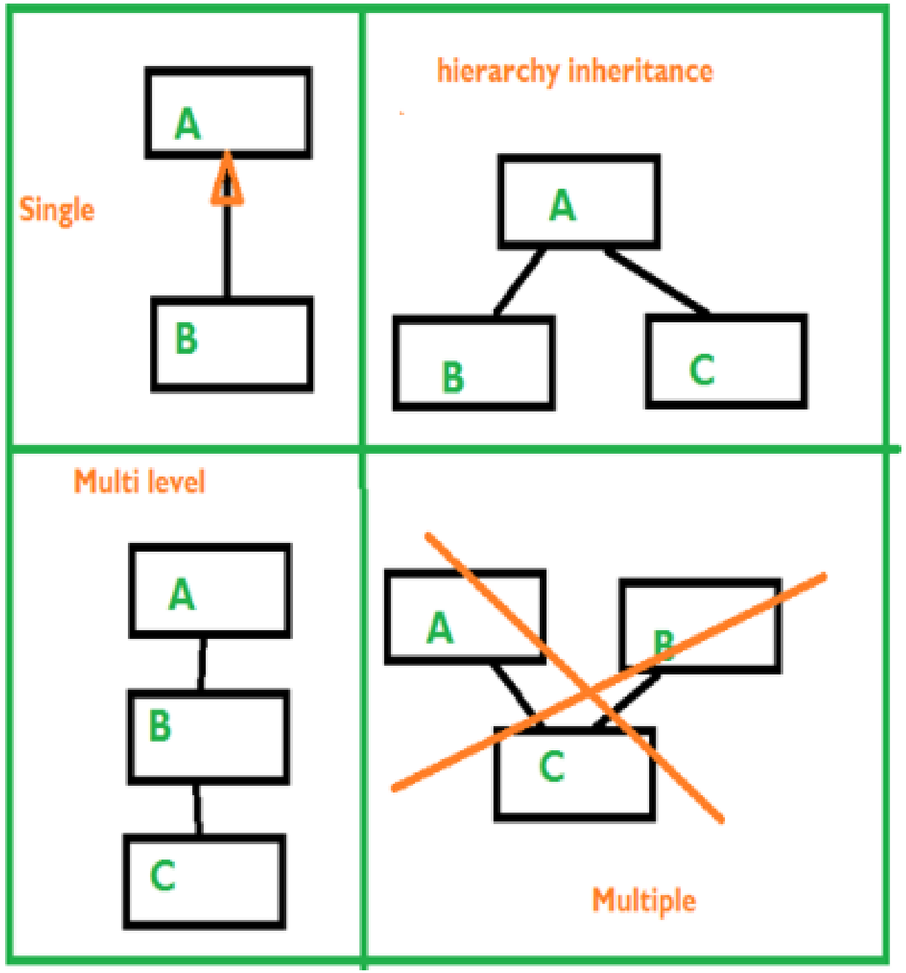




rgupta.mtech@gmail.com

Inheritance

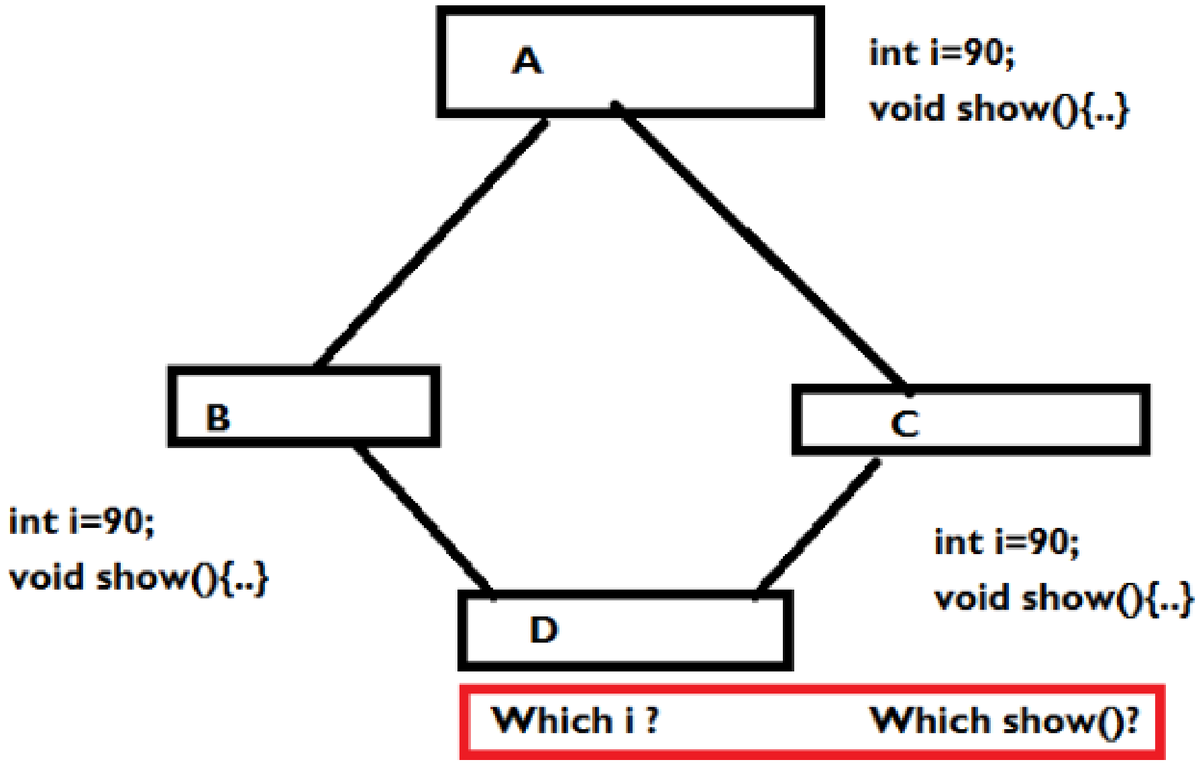
- **Inheritance is the inclusion of behaviour (i.e. methods) and state (i.e. variables) of a base class in a derived class so that they are accessible in that derived class.**
- **Code reusability.**
- **Subclass and Super class concept**



rgupta.mtech@gmail.com

Diamond Problem?

- **Hierarchy inheritance can leads to poor design..**
- **Java don't support it directly...(Possible using interface)**



rgupta.mtech@gmail.com

Inheritance example

- **Use extends keyword**
- **Use super to pass values to base class constructor.**

```
class A{
    int i;
    A()   {System.out.println("Default ctr of A");}

    A(int i) {System.out.println("Parameterized ctr of A");}
}

class B extends A{

    int j;
    B()   {System.out.println("Default ctr of B");}

    B(int i,int j)
    {
        super(i);
        System.out.println("Parameterized ctr of B");
    }

}
```

rgupta.mtech@gmail.com

Overloading

- **Overloading deals with multiple methods in the same class with the same name but different method signatures.**

```
class MyClass {
    public void getInvestAmount(int rate) {…}

    public void getInvestAmount(int rate, long principal)
    { … }
}
```

- **Both the above methods have the same method names but different method signatures, which mean the methods are overloaded.**
- **Overloading lets you define the same operation in different ways for different data.**
- **Constructor can be overloaded ambiguity**
- ***Overloading in case of var-arg and Wrapper objects…**

rgupta.mtech@gmail.com

Overriding...

- **Overriding deals with two methods, one in the parent class and the other one in the child class and has the same name and signatures.**
- **Both the above methods have the same method names and the signatures but the method in the subclass MyClass overrides the method in the superclass BaseClass**
- **Overriding lets you define the same operation in different ways for different object types.**

```
class BaseClass{
    public void getInvestAmount(int rate) {...}
}

class MyClass extends BaseClass {
    public void getInvestAmount(int rate) { ...}
}
```

rgupta.mtech@gmail.com

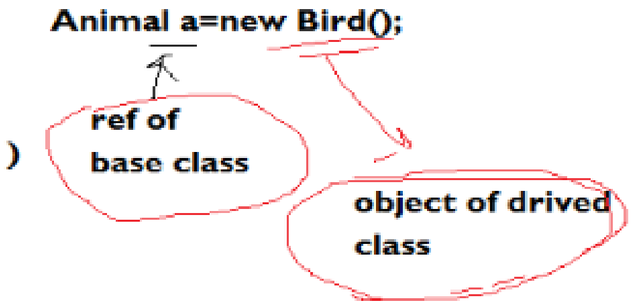
Polymorphism

- **Polymorphism=many forms of one things**
- **Substitutability**
- **Overriding**
- **Polymorphism means the ability of a single variable of a given type to be used to reference objects of different types, and automatically call the method that is specific to the type of object the**
- **variable references.**

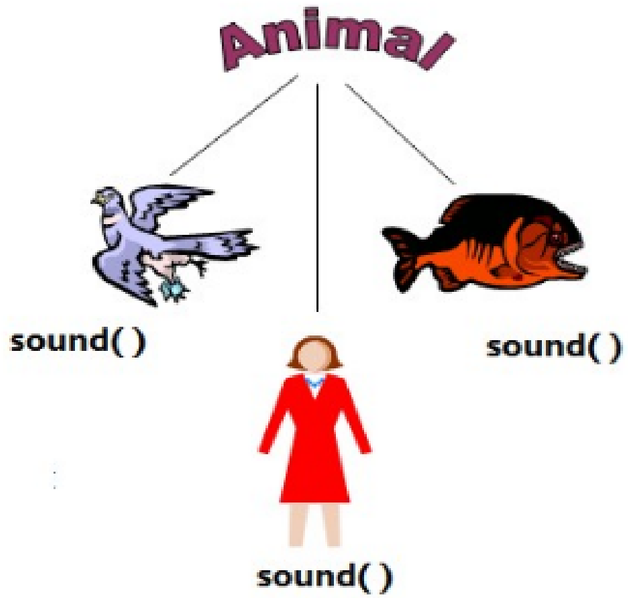
rgupta.mtech@gmail.com

Polymorphism

- **Every animal sound but differently…**
- **We want to have Pointer of Animal class assigned by object of derived class**



Which method is going is to be called is not decided by the type of pointer rather object assigned will decide at run time



rgupta.mtech@gmail.com

Example Polymorphism

```
class Animal
{
    public void sound()
    {
        System.out.println("Don't know how generic animal sound.....");
    }
}
class Bird extends Animal
{@Override
    public void sound()
    {
        System.out.println("Bird sound.....");
    }
}
class Person extends Animal
{@Override
    public void sound()
    {
        System.out.println("Person sound.....");
    }
}
```

rgupta.mtech@gmail.com

Need of abstract class?

- **Sound( ) method of Animal class don't make any sense**
- **...i.e. it don't have semantically valid definition**

- **Method sound( ) in Animal class should be abstract means incomplete**

- **Using abstract method Derivatives of Animal class forced to provide meaningful sound() method**

```
class Animal
{
    public void sound()
    {
        System.out.println("Don't know how generic animal sound.....");
    }
}
class Bird extends Animal
{@Override
    public void sound()
    {
        System.out.println("Bird sound.....");
    }
}
class Person extends Animal
{@Override
    public void sound()
    {
        System.out.println("Person sound.....");
    }
}
```

rgupta.mtech@gmail.com

Abstract class

- If an class have at least one abstract method it should be declare abstract class.
- Some time if we want to stop a programmer to create object of some class...
- Class has some default functionality that can be used as it is.
- Can extends only one abstract class

```
class Foo{
        public abstract void foo();
        }
```

```
abstract class Foo{
        public abstract void foo();
        }
```

```
abstract class Foo{
                }
```

rgupta.mtech@gmail.com

Abstract class use cases...

Want to have some default functionality from base class and class have some abstract functionality that can't be define at that moment.

```
abstract class Animal
{
        public abstract void sound();
        public void eat()
         {
                System.out.println("animal eat...");
          }
}
```

Don't want to allow a programmer to create object of an class as it is too generic

Interface vs abstract class

rgupta.mtech@gmail.com

Example Abstract class

```
public abstract class Account  {
    public void deposit (double amount) {
        System.out.println("depositing " + amount);
    }

    public void withdraw (double amount) {
        System.out.println ("withdrawing " + amount);
    }

    public abstract double calculateInterest(double  amount);
}
```

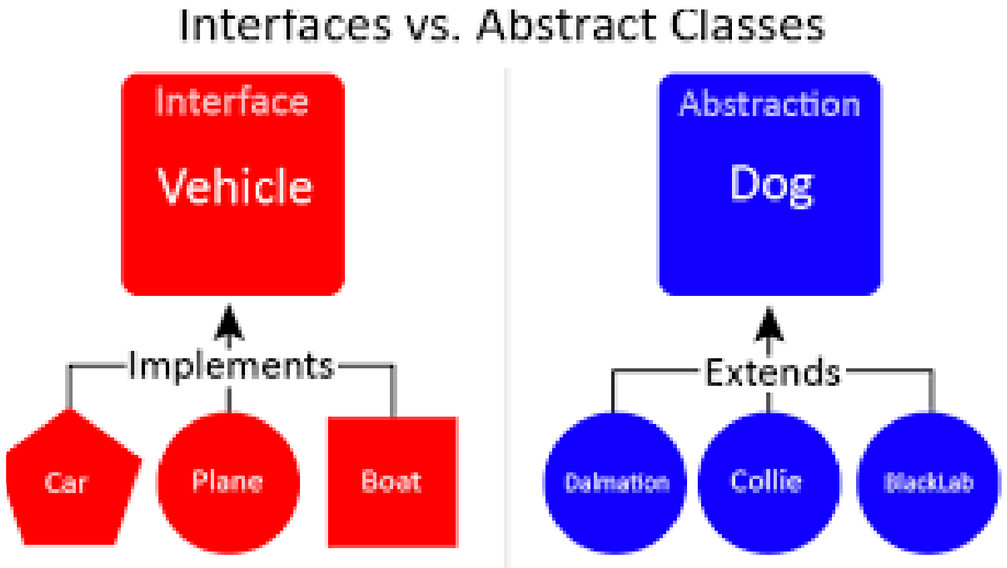
```
public class SavingsAccount extends Account {

    public double calculateInterest (double amount) {
        // calculate interest for SavingsAccount
        return amount * 0.03;
    }

    public void deposit (double amount) {
        super.deposit (amount); // get code reuse
        // do something else
    }

    public void withdraw (double amount) {
        super.withdraw (amount); // get code reuse
        // do something else
    }
}
```

```
public class TermDepositAccount extends Account {

    public double calculateInterest (double amount) {
        // calculate interest for SavingsAccount
        return amount * 0.05;
    }

    public void deposit(double amount) {
        super.deposit (amount); // get code reuse
        // do something else
    }

    public void withdraw(double amount) {
        super.withdraw (amount); // get code reuse
        // do something else
    }
}
```

rgupta.mtech@gmail.com

Interface vs Abstract class

	Interface	Abstract Class
Constructors	✗	✓
Static Fields	✓	✓
Non-static Fields	✗	✓
Final Fields	✓	✓
Non-final Fields	✗	✓
Private Fields & Methods	✗	✓
Protected Fields & Methods	✗	✓
Public Fields & Methods	✓	✓
Abstract methods	✓	✓
Static Methods	✓	✓
Final Methods	✗	✓
Non-final Methods	✓	✓
Default Methods	✓	✗



rgupta.mtech@gmail.com

```
interface Foo{
        void foo();
}
```

```
interface Foo{
  public abstract void foo();

}
```

```
interface Foo{
        protected void foo();
}
```

final

```
interface Foo{
        int i=9;
}
```

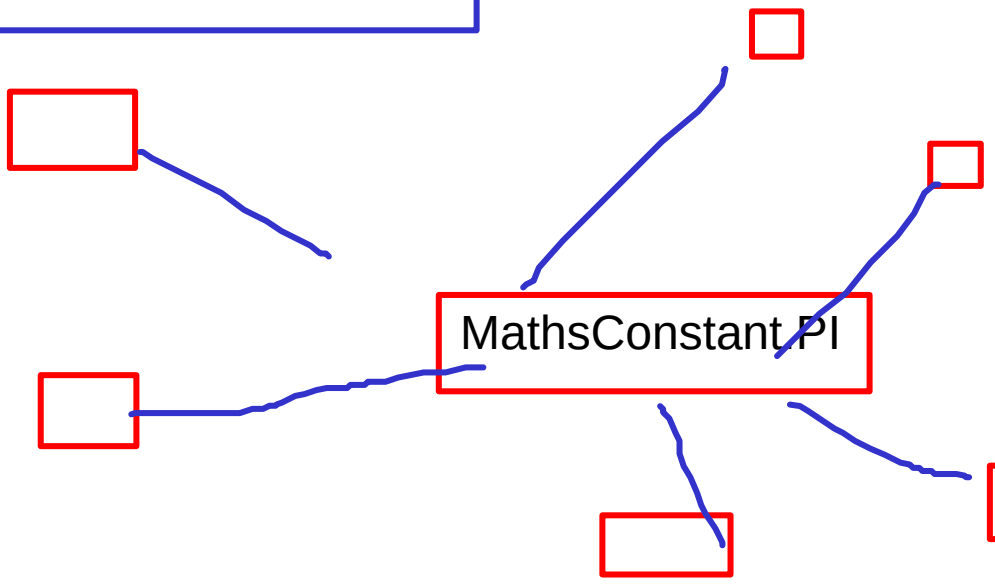
```
interface Foo{
        public static final int i=49;
}
```

U can not put instance variable inside the interface:
interface can not have ctr

we can not define a variable inside a interface (who can very...)

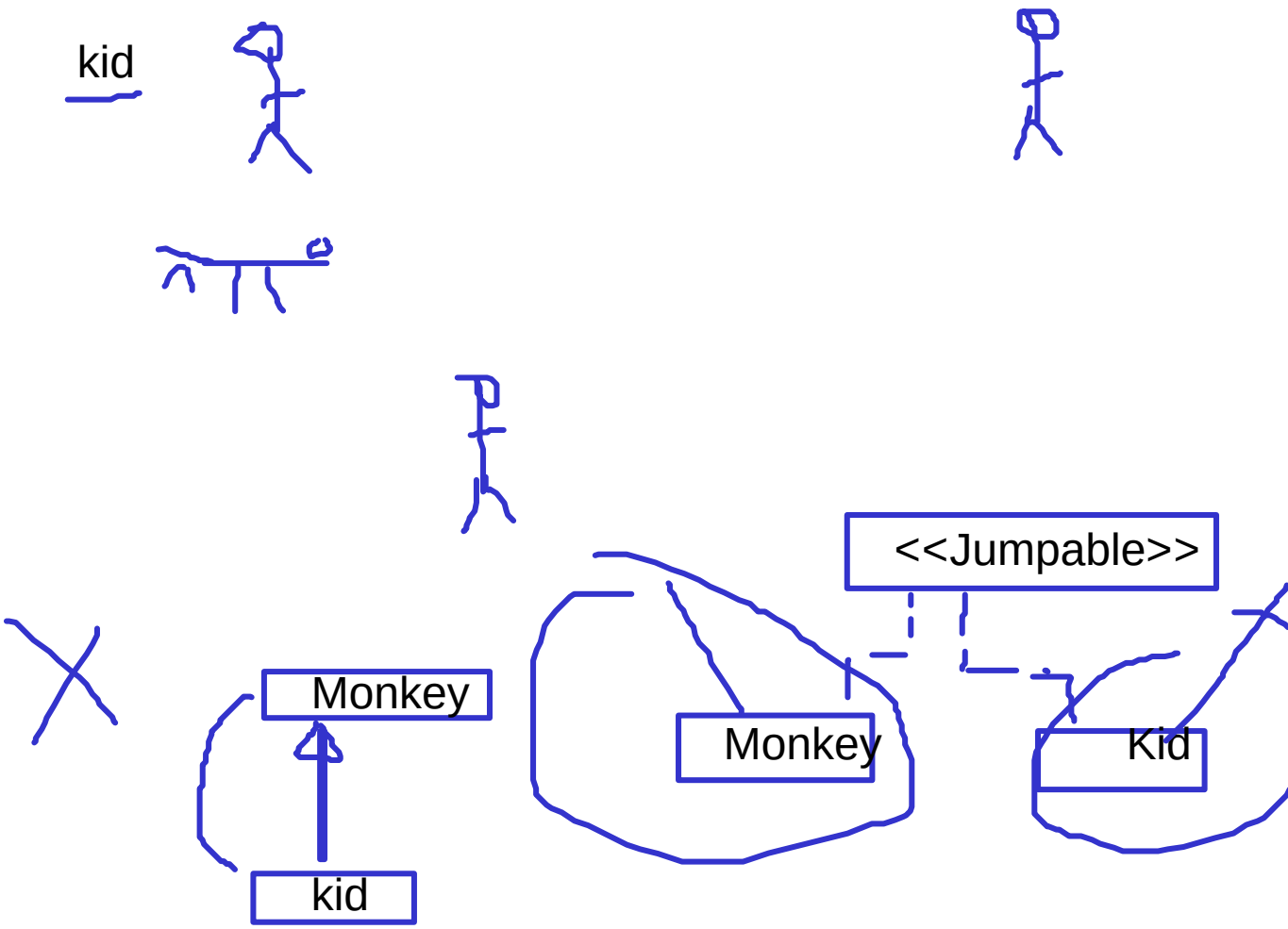
we can only define aka global constructor

```
interface MathsConstant{

        double PI=3.14;
}
```



enum: java 5
special class

used to defined
named
constant


kid
<<Jumpable>>
Monkey
kid
Monkey
Kid

Interface vs Abstract class

Interface: Specification of a behavior . The interface represents some features of a class

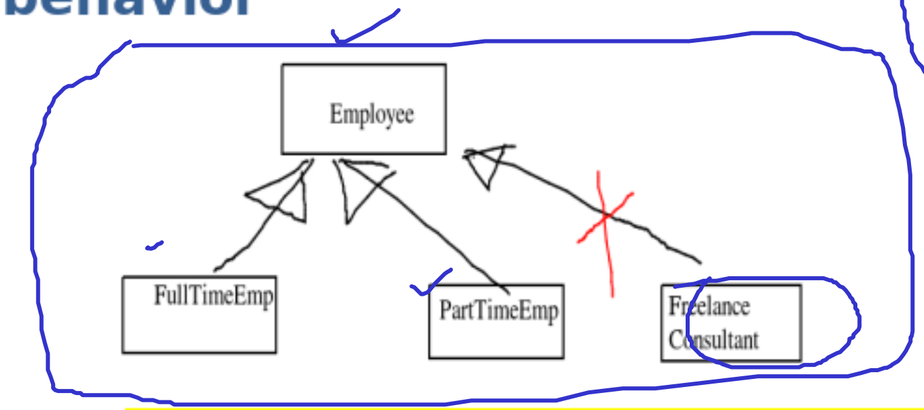
Abstract class: generalization of a behavior The incomplete class that required further specialization

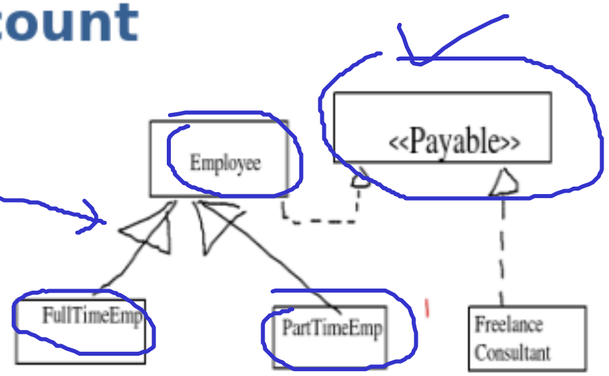
What an object can do?

What an object is?

Standardization of a behavior

IS-A SavingAccount IS-A Account



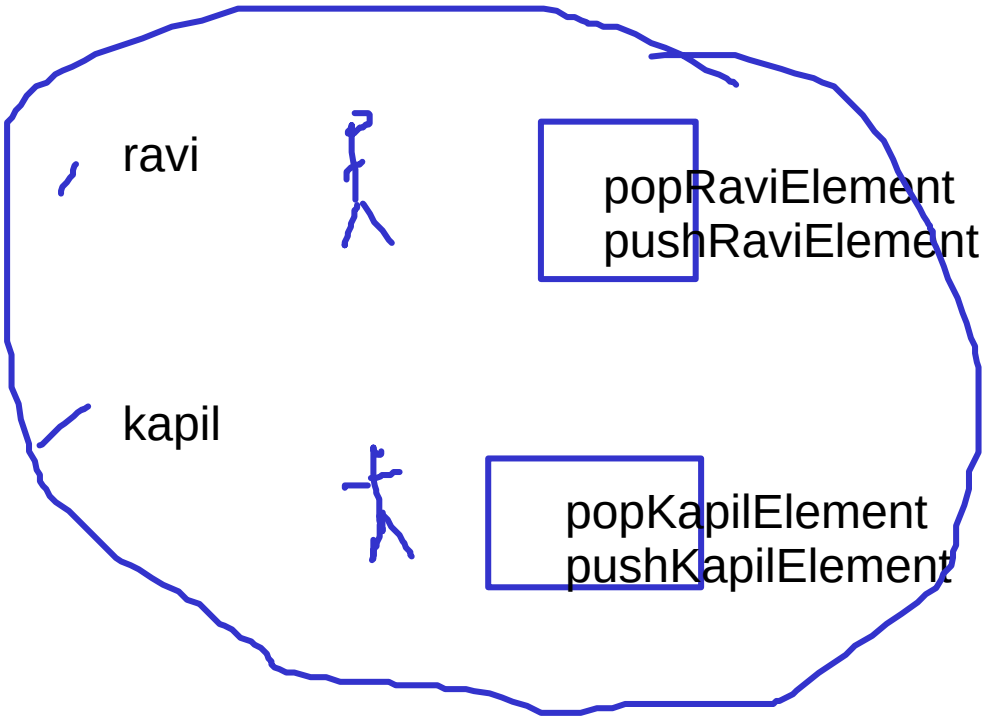


When we should go for interface and when we should go for abstract class?
Interface break the hierarchy

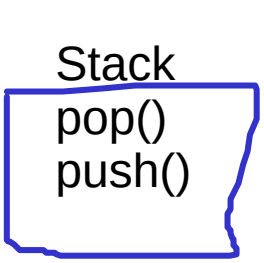
rgupta.mtech@gmail.com

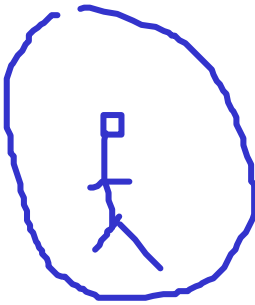
how interface help us

act as a contract bw 2 parties
or
it is a specification of a behaviour

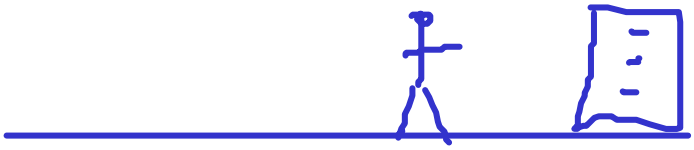


jr dev

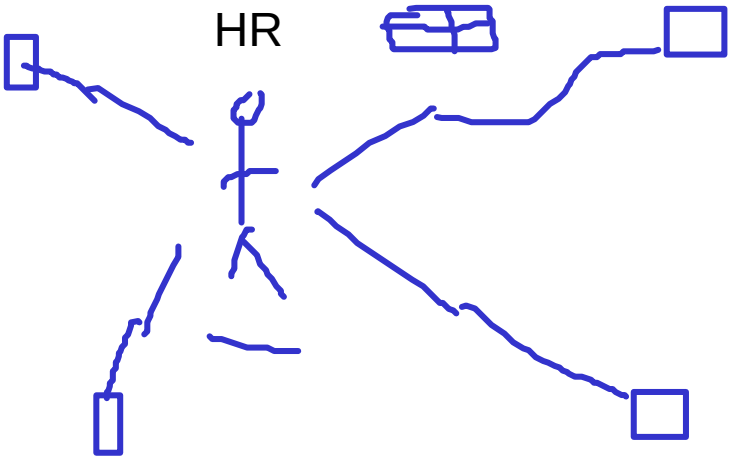




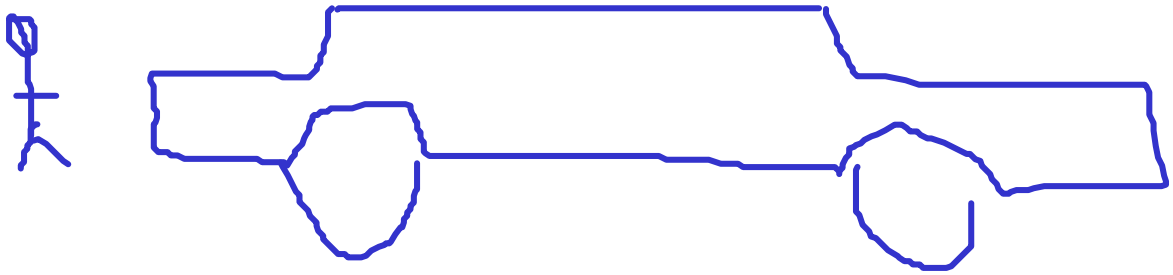
me



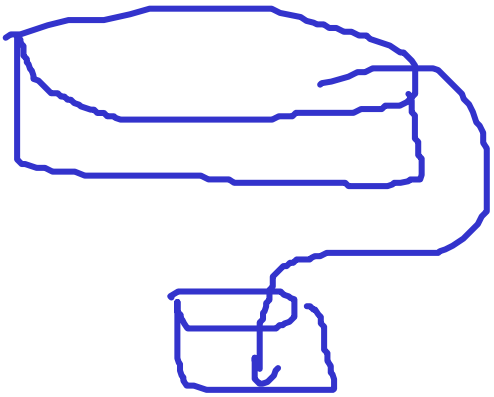
HR



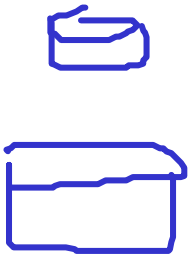
Upcating and downcasting?

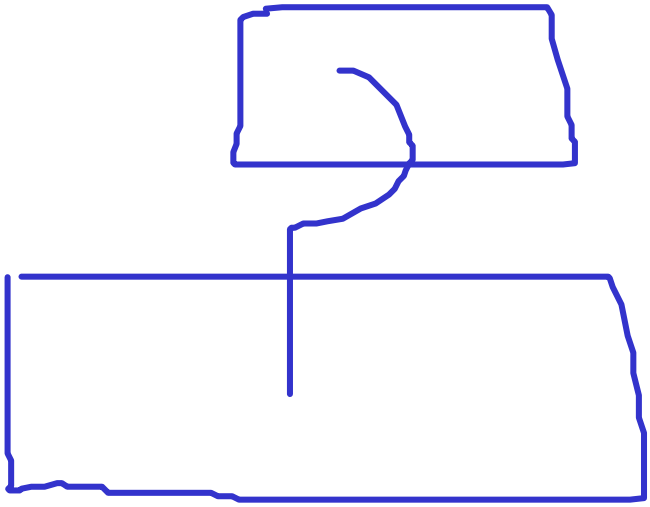


```
double a=6;
int i=a;
```

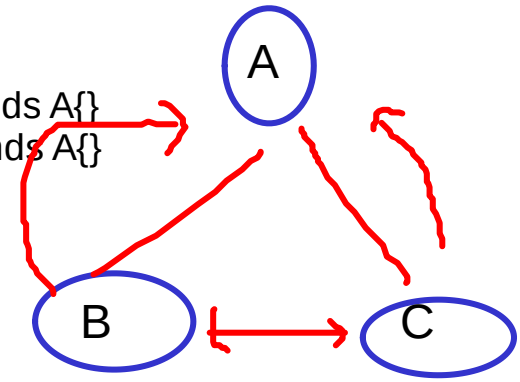


```
int i=58687;
      long k= i;//upcasting , plesent
      System.out.println(k);
```





```
class A{

}
class B extends A{}
class C extends A{}
```



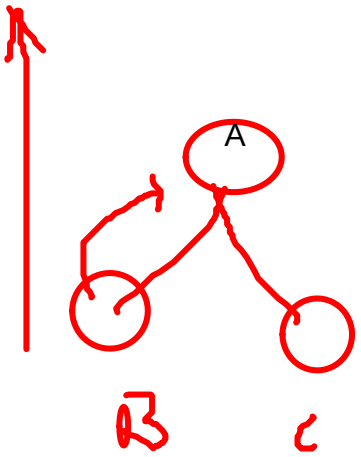
A a=new B();

A a=new A();

A a=new B(); //upcasting

```
class A{
      void foo(){
         System.out.println("foo of base class A");
      }
}
class B extends A{
   void fooB(){
      System.out.println("fooA of base class B");
   }
}
class C extends A{
   void fooC(){
      System.out.println("fooC of base class C");
   }
}
public class UpcastingVsDowncatingForObjects {
   public static void main(String[] args) {
     A a=new B();//upcasting , going towards top of heriar....

   B b=new A();//downcasting
   b.fooB();


}
}
```

```
class A{
    void foo(){
        System.out.println("foo of base class A");
    }
}
class B extends A{
    void fooB(){
        System.out.println("fooB of base class B");
    }
}
class C extends A{
    void fooC(){
        System.out.println("fooC of base class C");
    }
}
public class UpcastingVsDowncatingForObjects {
    public static void main(String[] args) {
        B b=new B();

        A a=b;//using ref of A u can only call method witch are also there in base class

        a.foo();

        //if u want to use exclusive method of class B u have use typecasting
        ((B) a).fooB();


    }
}
```

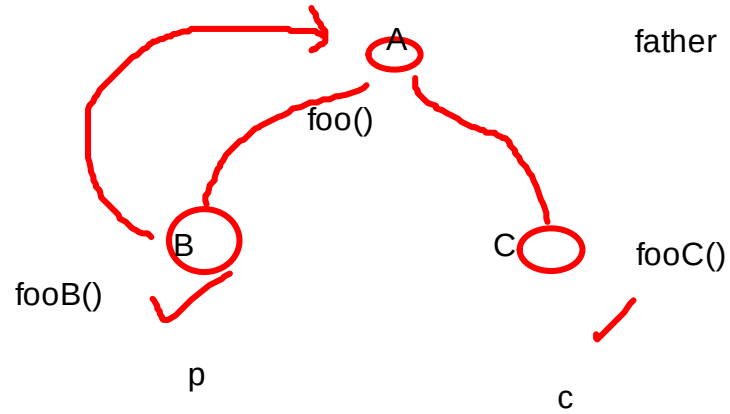


ClassCast Exception

listen pointer is pointing to object of B
(i force it)

```
B b=new B();
A a=b;
((C) a).fooC();
```

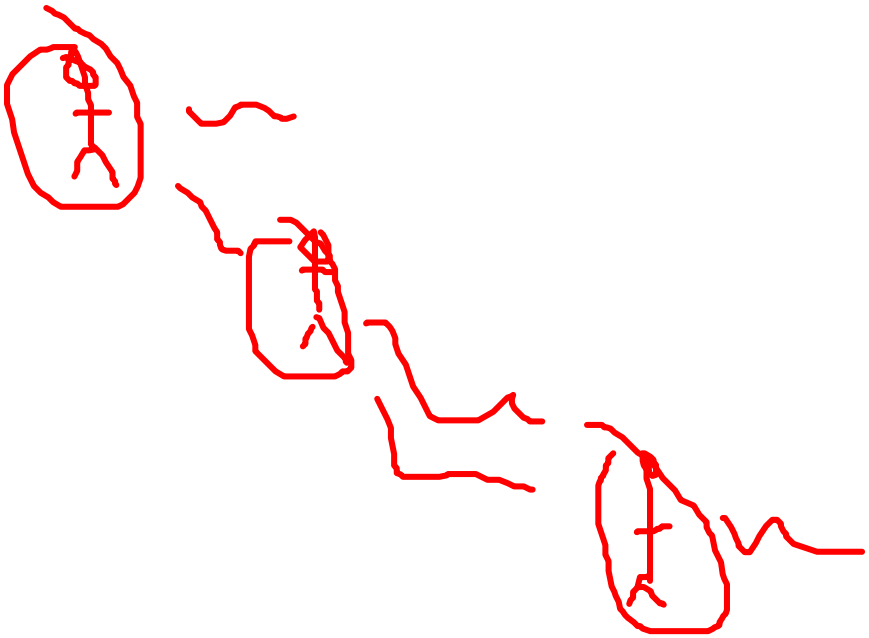
CCEx




```
Exception in thread "main" java.lang.ClassCastException:
 com.day3.session1.ex5.interface_vs_abs.d.B cannot be cast
to com.day3.session1.ex5.interface_vs_abs.d.C
        at com.day3.session1.ex5.interface_vs_abs.d.Upcasting
VsDowncatingForObjects.main(UpcastingVsDowncatingForObjects.java:21)

Process finished with exit code 1
```

```
Animal [] animals=getAnimals();
    //if animal is dog then ask him to do nightWatchManShip otherwise ask him to sound

    for(Animal a: animals){
       a.sound();
       if(a instanceof Dog)
          ((Dog)a).nightWatchManShip();
    }
```

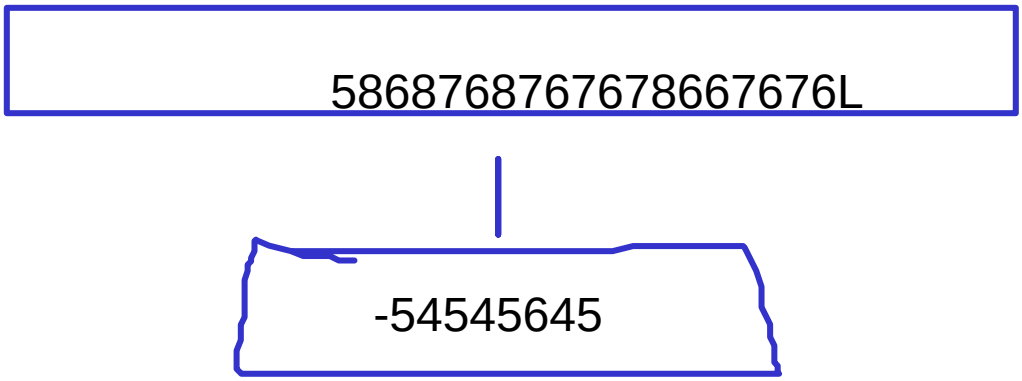
```
public static Animal[] getAnimals() {
        Animal [] animals={new Dog(), new Dog(), new Cat(), new Cat(), new Dog()};
        return animals;
    }
```



- long i=586876876767866767 6L;

int k= (int) i;//downcating , forcefully

System.out.println(k);



final keyword

- What is the meaning of final
  - Something that can not be change!!!
- final
  - Final method arguments
    - Cant be change inside the method
  - Final variable
    - Become constant, once assigned then cant be changed
  - Final method
    - Cant overridden
  - Final class
    - Can not inherited (Killing extendibility)
    - Can be reuse

Some examples….

rgupta.mtech@gmail.com

Final class

Final class can’t be subclass i.e. Can’t be extended
No method of this class can be overridden
Ex: String class in Java…

Real question is in what situation somebody should declare a class final

```
package cart;

public final class Beverage{

     public void importantMethod(){
          Sysout("hi");
     }
}
```

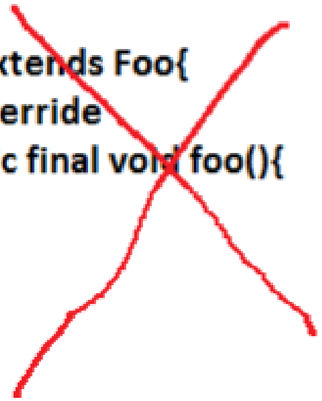
```
package examStuff;
import cart.*;

class Tea extends beverage{

}
```

rgupta.mtech@gmail.com

Final Method

- **Final Method Can’t overridden**
- **Class containing final method can be extended**
- **Killing substitutability**

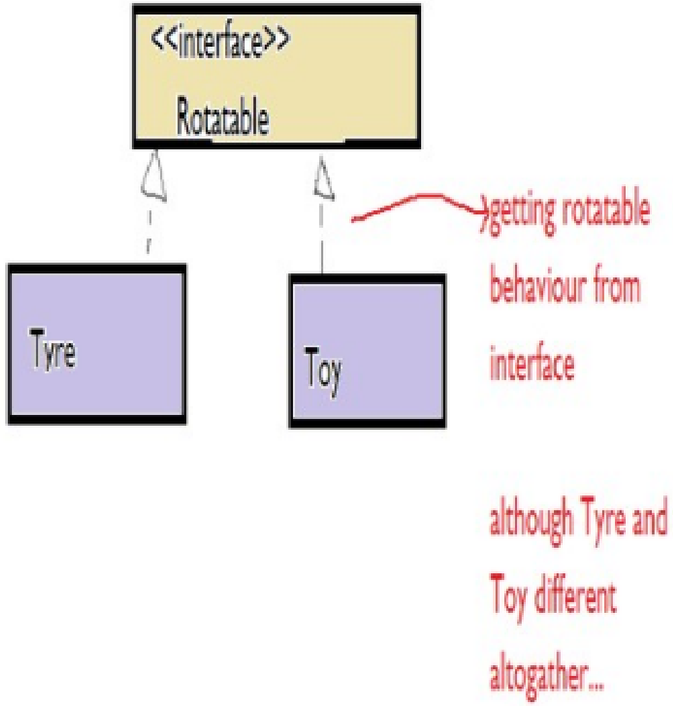
```
class Foo{

        public final void foo(){
                Sysout("i am the best");
                Sysout("You can use me but can't override me");
        }
};

class Bar extends Foo{
        @Override
        public final void foo(){


        }
}
```



rgupta.mtech@gmail.com

Interface?

- **Interface : Contract bw two parties**
- **Interface method Only declaration**
- **No method definition**
- **Interface variable are Public static and final constant**
- **Its how java support global constant Break the hierarchy**
- **Solve diamond problem**
- **Callback in Java**



rgupta.mtech@gmail.com

Implementing an interface

What we declare

```
interface Bouncable{
      int i=9;
     void bounce();
     void setBounceFactor();
}
```

What compiler think...

```
interface Bouncable{
     public static final int i=9;
     public abstract void bounce();
     public abstract void setBounceFactor();
}
```

All interface method must be implemented....

```
class Tyre implements Bouncable{

     public void bounce(){
          Sysout(i);
          Sysout(i++);
     }
     public void setBounceFactor(){}
}
```

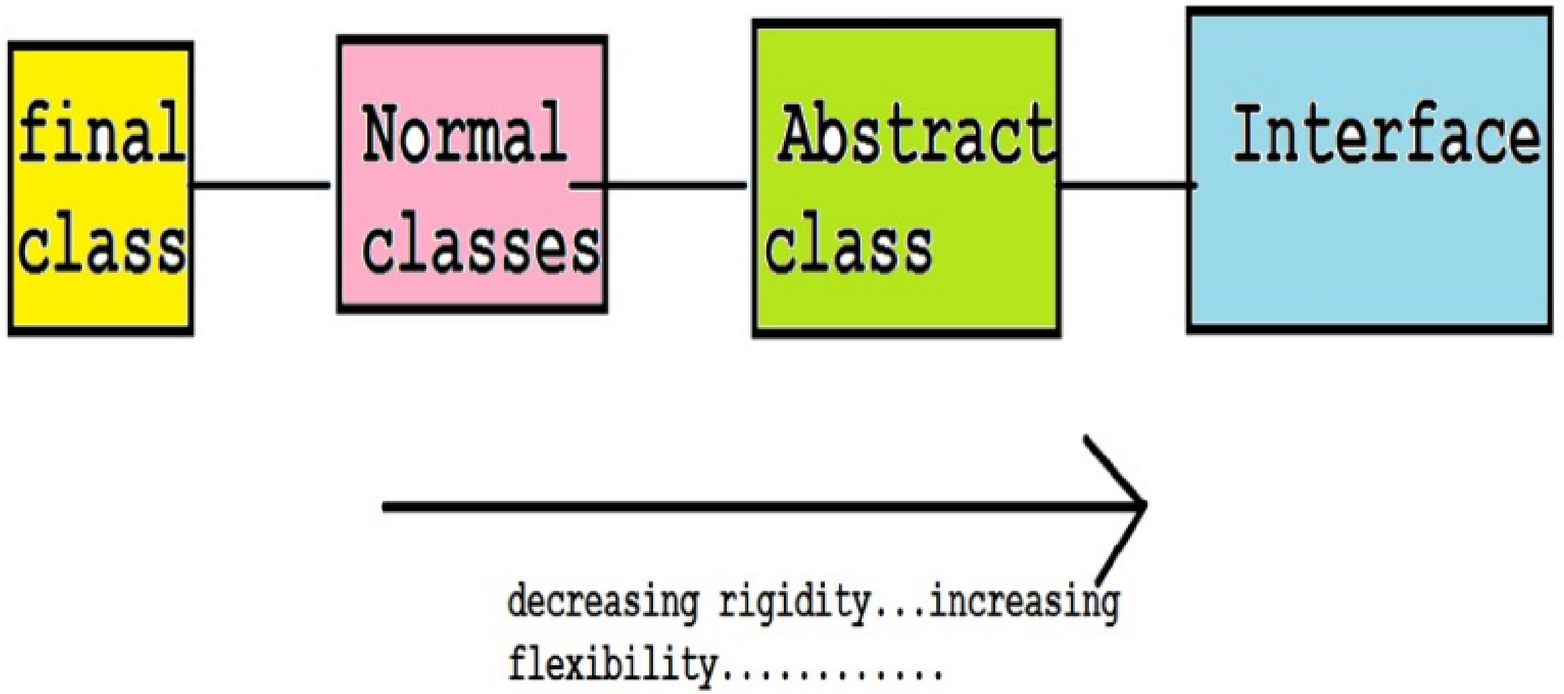
rgupta.mtech@gmail.com

Note on interface

- **Following interface constant declaration are identical**
  - **int i=90;**
  - **public static int i=90;**
  - **public int i=90;**
  - **public static int i=90;**
  - **public static final int i=90;**
- **Following interface method declaration don’t compile**
  - **final void bounce();**
  - **static void bounce();**
  - **private void bounce();**
  - **protected void bounce();**

rgupta.mtech@gmail.com

Decreasing Rigidity..increasing Flexibility



rgupta.mtech@gmail.com

Type of relationship bw objects

- USE-A
- HAS-A
- IS-A (Most costly ?)

ALWAYS GO FOR LOOSE COUPLING AND HIGH COHESION...
But HOW?

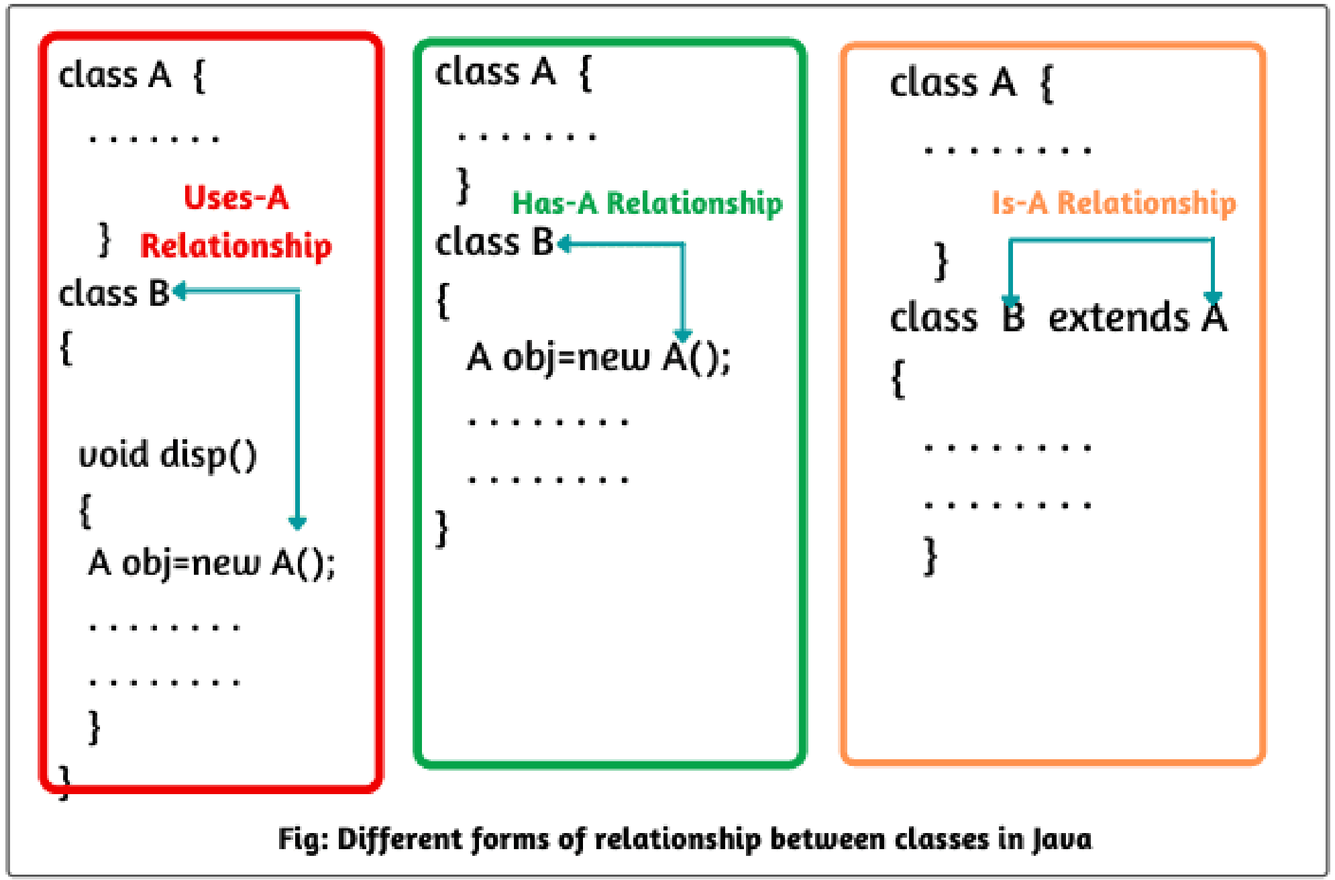
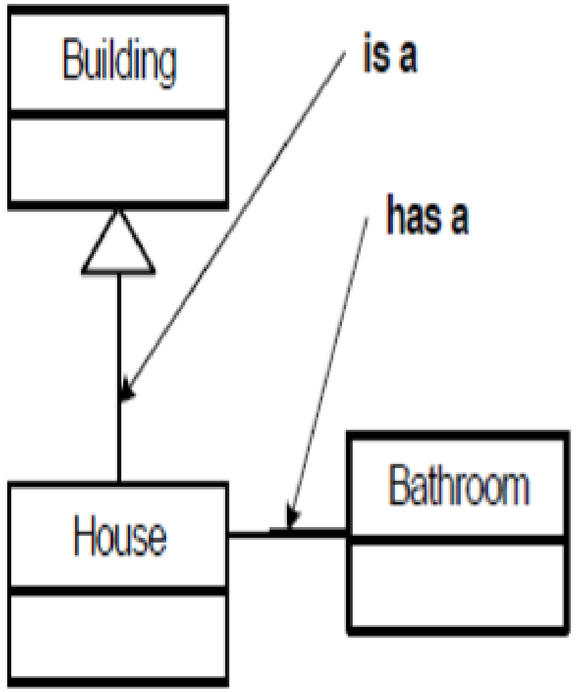


Fig: Different forms of relationship between classes in Java

rgupta.mtech@gmail.com

Inheritance vs Composition

Inheritance [is a] Vs Composition [has a]



is a [House is a Building]

```
class Building{
 .......
}

class House extends Building{
 .........
}
```

has a [House has a Bathroom]

```
class House {
    Bathroom room = new Bathroom() ;
    ....
   public void getTotMirrors(){
       room.getNoMirrors();
       ....
   }
}
```

rgupta.mtech@gmail.com

Session 4:
String, Wrapper classes, Immutability
Inner classes, Java 5 features

rgupta.mtech@gmail.com

String

Immutable i.e. once assigned then can’t be changed

Only class in java for which object can be created with or without using new operator

Ex: String s=“india”;

String s1=new String(“india”);

What is the difference?

String concatenation can be in two ways:

String s1=s+ “paki”; Operator overloading

String s3=s1.concat(“paki”);

rgupta.mtech@gmail.com

Immutability

- **Immutability means something that cannot be changed.**
- **Strings are immutable in Java. What does this mean?**
- **String literals are very heavily used in applications and they also occupy a lot of memory.**
- **Therefore for efficient memory management, all the strings are created and kept by the JVM in a place called string pool (which is part of Method Area).**
- **Garbage collector does not come into string pool. How does this save memory?**

RULES FOR IMMUTABILITY

Declare the class as final so it can't be extended.

Make all fields private so that direct access is not allowed.

Don't provide setter methods for variables

Make all mutable fields final so that it's value can be assigned only once.

Initialize all the fields via a constructor performing deep copy.

Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

rgupta.mtech@gmail.com

How to make copy of an Object?

```
int[] src  = new int[]{1,2,3,4,5};
int[] dest = new int[5];
System.arraycopy( src, 0, dest, 0, src.length );
```

```
int[] a = new int[]{1,2,3,4,5};
int[] b = a.clone();
```

```
int[] a = {1,2,3,4,5};
int[] b = Arrays.copyOf(a, a.length);
```

Arrays.copyOfRange():
If you want few elements of an array to be copied

```
public final class ImmutableReminder{
    private final Date remindingDate;

    public ImmutableReminder (Date remindingDate) {
        if(remindingDate.getTime() < System.currentTimeMillis()){
            throw new IllegalArgumentException("Can not set reminder"
                    " for past time: " + remindingDate);
        }
        this.remindingDate = new Date(remindingDate.getTime());
    }

    public Date getRemindingDate() {
        return (Date) remindingDate.clone();
    }
}
```

rgupta.mtech@gmail.com

Rules for creating immutable class

There are certain steps that are to be followed for creating an immutable class –

1. Methods of the class should not be overridden by the subclasses. You can ensure that by making your class **final**.
2. Make all **fields final and private**. If the field is of **primitive type**, then it's value can't be changed as it is final. If field is holding a reference to another object, then declaring that field as final means its reference can't be changed.
   Having **access modifier** as private for the fields ensure that fields are not accessed outside the class.
3. Initialize all the fields in a **constructor**.
4. Don't provide setter methods or any method that can change the state of the object. Only provide methods that can access the value (like getters).
5. In case any of the fields of the class holds reference to a mutable object any change to those objects should also not be allowed, for that –
   - Make sure that there are no methods within the class that can change those mutable objects (change any of the field content).
   - Don't share reference of the mutable object, if any of the methods of your class return the reference of the mutable object then its content can be changed.
   - If reference must be returned create copies of your internal mutable objects and return those copies rather than the original object. The copy you are creating of the internal mutable object must be a **deep copy** not a shallow copy.

rgupta.mtech@gmail.com

Some common string methods...

String	methods
charAt()	Returns the character located at the specified index
concat()	Appends one String to the end of another ("+" also works)
equalsIgnoreCase()	Determines the equality of two Strings, ignoring case
length()	Returns the number of characters in a String
replace()	Replaces occurrences of a character with a new character
substring()	Returns a part of a String
toLowerCase()	Returns a String with uppercase characters converted
toString()	Returns the value of a String
toUpperCase()	Returns a String with lowercase characters converted
trim()	Removes whitespace from the ends of a String

rgupta.mtech@gmail.com

String comparison

Two string should never be checked for equality using == operator WHY?
Always use equals( ) method....

String s1="india"; String s2="paki";

if(s1.equals(s2))
.....
.....

rgupta.mtech@gmail.com

SOLID Principles

	String	StringBuffer	StringBuilder
Storage Area	Constant String Pool	Heap	Heap
Modifiable	No (immutable)	Yes(mutable)	Yes(mutable)
Thread Safe	Yes	Yes	No
Performance	Fast	Very slow	Fast

```
Various memory area of JVM
    1. Method area          ------ Per JVM
    2. heap area            ------ Per JVM
    3. stack area           ------ Per thread
    4. PC register area     ------ Per thread
    5. Native method area   ------ Per thread
```

String	StringBuffer / StringBuilder (added in J2SE 5.0)
String is **immutable**: you can't modify a string object but can replace it by creating a new instance. Creating a new instance is rather expensive.	StringBuffer is **mutable**: use StringBuffer or StringBuilder when you want to modify the contents. ***StringBuilder*** was added in Java 5 and it is identical in all respects to ***StringBuffer*** except that it is not synchronized, which makes it slightly faster at the cost of not being thread-safe.

//Inefficient version using immutable String
String output = "Some text"
Int count = 100;
for(int i =0; i<count; i++) {
output += i;
}
return output;
	//More efficient version using mutable StringBuffer
	StringBuffer output = new **StringBuffer**(**110**);// set an initial size of 110
	output.append("Some text");
	for(int i =0; i<count; i++) {
	output.append(i);
	}
	return output.toString();

The above code would build 99 new String objects, of which 98 would be thrown away immediately. Creating new objects is not efficient.	The above code creates only two new objects, the *StringBuffer* and the final *String* that is returned. StringBuffer expands as needed, which is costly however, so it would be better to initialize the *StringBuffer* with the correct size from the start as shown.

rgupta.mtech@gmail.com

Wrapper classes

- Helps treating primitive data as an object
- But why we should convert primitive to objects?
  - We can’t store primitive in java collections
  - Object have properties and methods
  - Have different behavior when passing as method argument
- Eight wrapper for eight primitive
- Integer, Float, Double, Character, Boolean etc…
- Integer it=new Integer(33); int temp=it.intValue();
- ….

primitive==>object

```
Integer it=new Integer(i);
```

object==>primitive

```
int i=it.intValue()
```

primitive ==>string

```
String s=Integer.toString();
```

String==>Numeric object

```
Double val=Double.valueOf(str)
```

rgupta.mtech@gmail.com

Boxing / Unboxing Java 1.5

Boxing

```
Integer iWrapper = 10;  Prior to J2SE 5.0, we use
Integer a = new Integer(10);
```

Unboxing

```
int  iPrimitive = iWrapper;
Prior to J2SE 5.0, we use
int  b = iWrapper.intValue();
```

rgupta.mtech@gmail.com

Java 5 Language Features (I)

- **AutoBoxing and Unboxing**
- **Enhanced For loop**
- **Varargs**
- **Covariant Return Types**
- **Static Import**
- **Enums**

rgupta.mtech@gmail.com

Enhanced For loop

Provide easy looping construct to loop through array and collections

```
int x[]=new int[5];
....
....
for(int temp:x)
        Sysout("temp:"+temp);

class Animal{ }
class Cat extends Animal{   }
class Dog extends Animal{  }

Animal []aa={new Cat(),new Dog(), new Cat()};

for(Animal a:aa)
        .......
```

rgupta.mtech@gmail.com

Varargs

Java start supporting variable argument method in Java 1.5

```
class Foo{

        public void foo(int ...j)
        {
                for(int temp:j)
                        Sysout(temp);
        }
        .....
        .....
}
```

Discuss function overloading in case of Varargs and wrapper classes

rgupta.mtech@gmail.com

Static Import

- **Handy feature in Java 1.5 that allow something like this: import static java.lang.Math.PI;**

import static java.lang.Math.cos;

Now rather then using

double r = Math.cos(Math.PI * theta);

We can use something like

double r = cos(PI * theta); – looks more readable ….

rgupta.mtech@gmail.com

Enums

- **Enum is a special type of classes.**
- **enum type used to put restriction on the instance values**

```
public enum myCars{
    HONDA,BMW,TOYOTA
};
```

its optional !!!

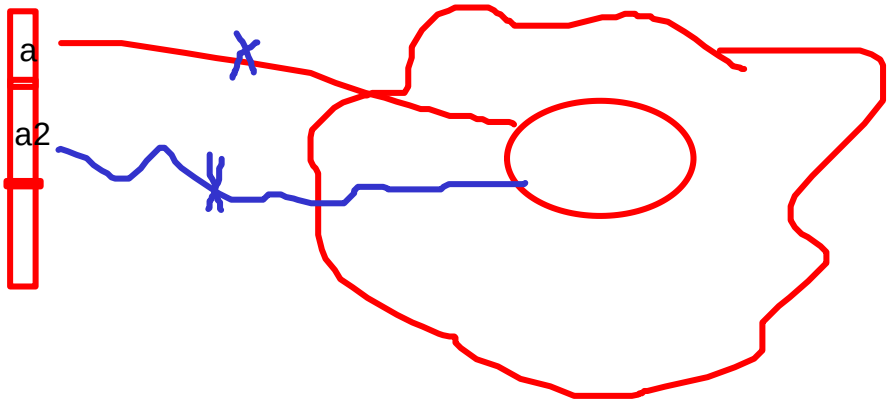
```
        ........
        ........
    myCars currentcar=myCars.BMW;
     System.out.println("My Current Cars : "+ myCars.BMW);
```

```
enum Day {
        SUNDAY, MONDAY, TUESDAY,WEDNESDAY, THURSDAY,
        FRIDAY, SATURDAY
}

//used
Day today = Day.WEDNESDAY;
switch(today){
        case SUNDAY:
                break;
        //...
}
```

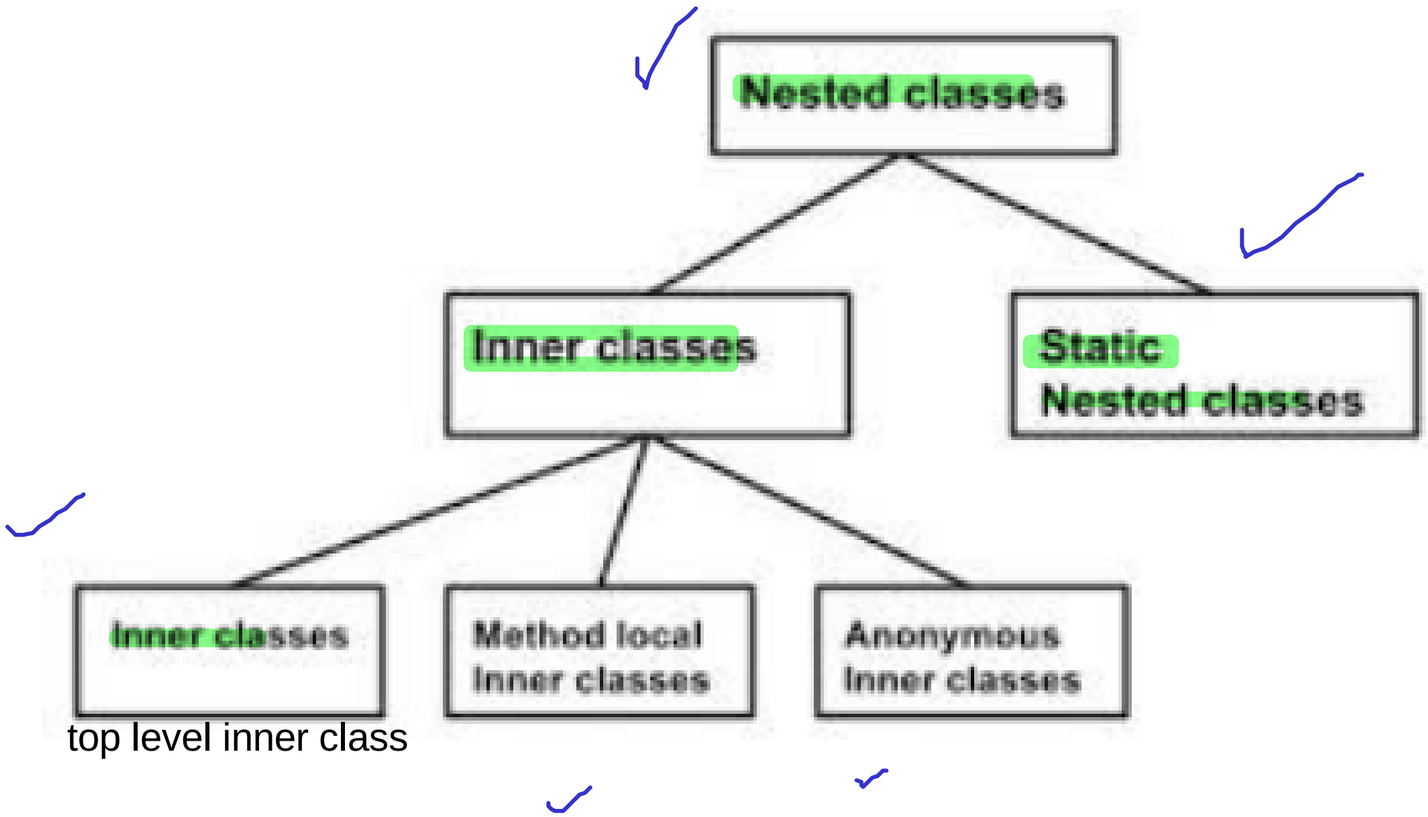
rgupta.mtech@gmail.com

```
class Milk{}
class PasteurizedMilk extends Milk{}

class KrishnaDairy{
   public Milk sellMilk(){
      return new Milk();
   }
}
//return type of drived class is a specilized type of base class => coverient return type
class ImprovedKrishnaDairy extends KrishnaDairy{
   @Override
   public PasteurizedMilk sellMilk(){
      return new PasteurizedMilk();
   }
}

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

   }
}
```

```
M a=new M();
    M a2=a;

    System.out.println(a.hashCode());
    System.out.println(a2.hashCode());
```



Inner classes



Inner class?

A class defined inside another class.

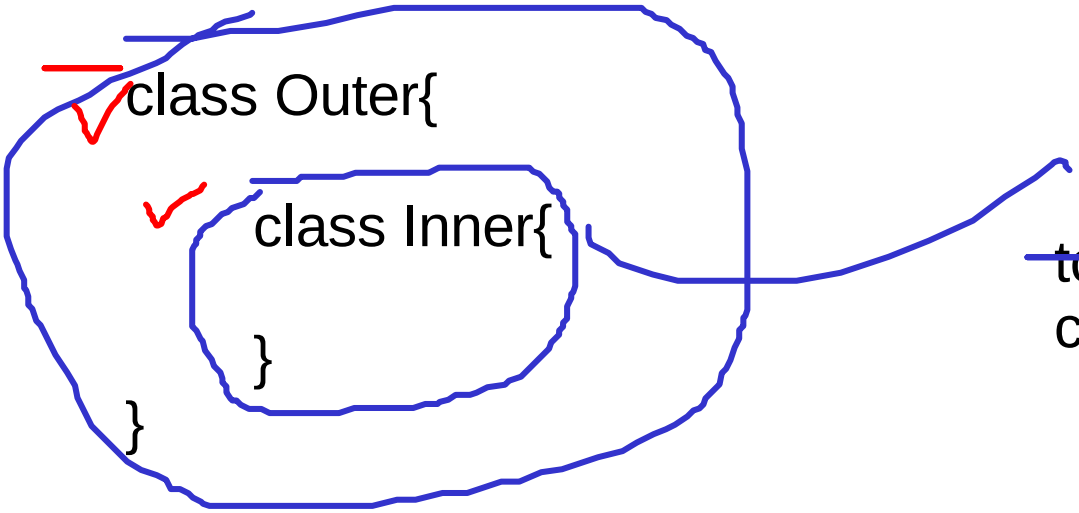
Why to define inner classes?

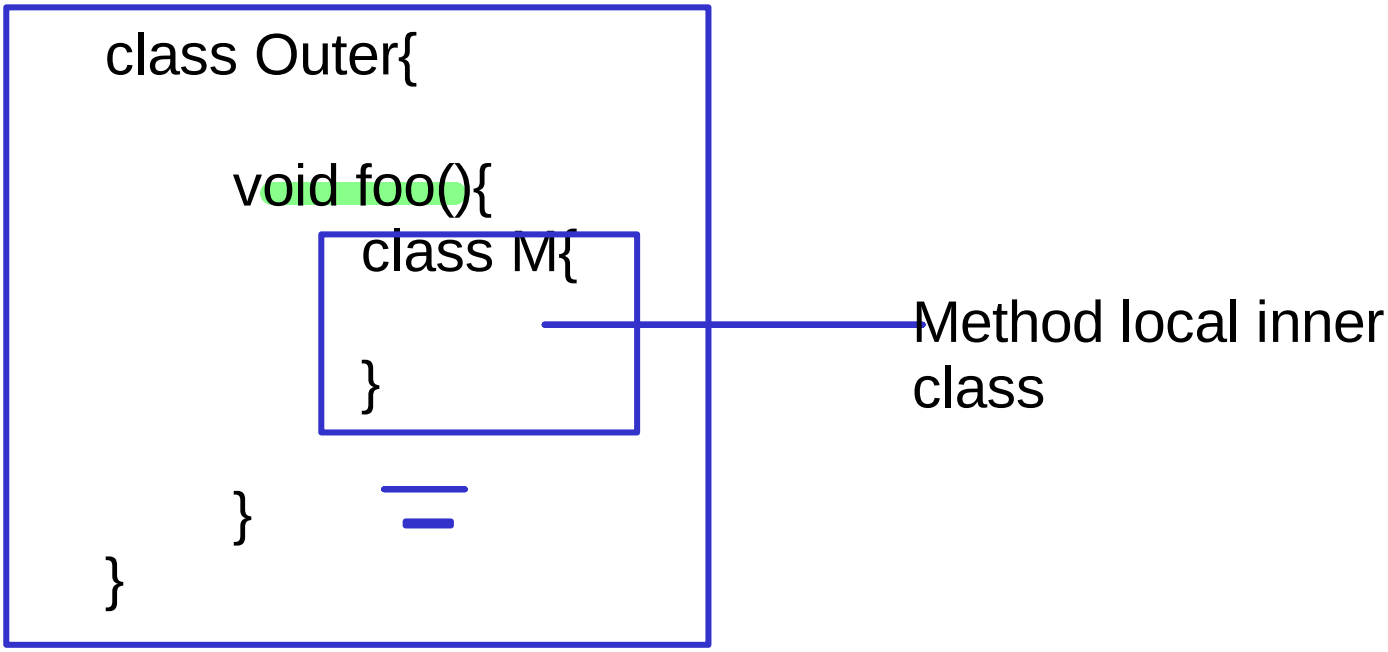
Inner classes

- **Top level inner class**
- **Method local inner class**
- **Anonymous Inner class**
- **Method local argument inner class**

Static inner classes

rgupta.mtech@gmail.com





imp

lambada expression

Annonymous inner class

CLass without name?

Object of Inner class can not be created without the object of outer class?

```
interface Cookable{
    void cook();
}
```

```
Cookable c=new Cookable() {
        @Override
        public void cook() {
            System.out.println("street food");
        }
```

limitation:

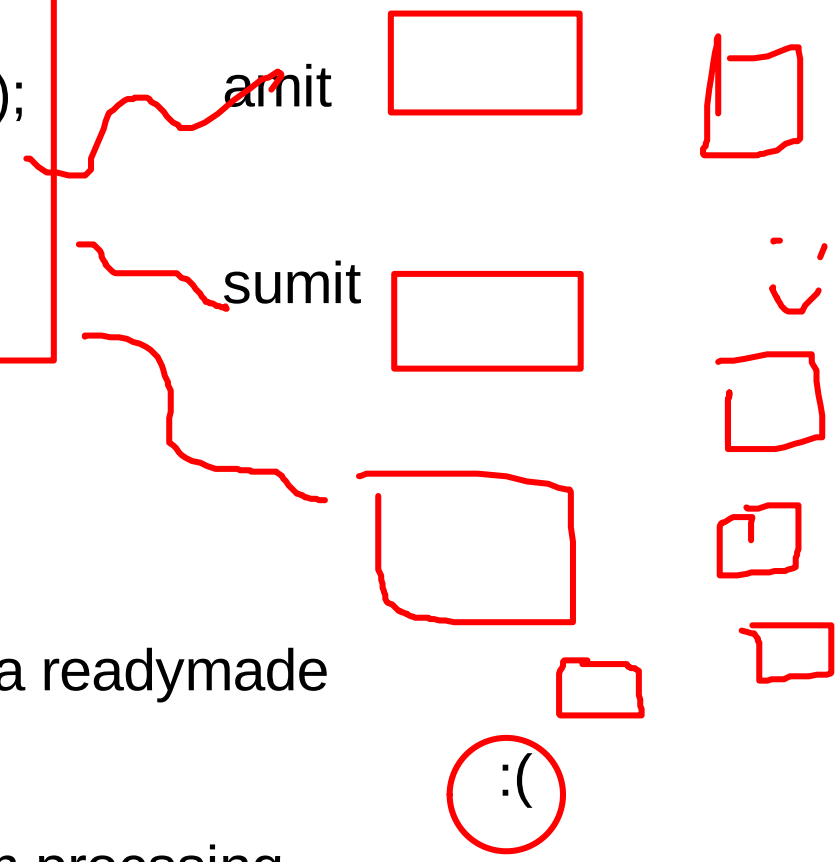
java 7 interface was 100% abstraction

rajeev

```
interface stack{
        void push(int i);
        int pop();
        int peek()

}
```

amit

sumit



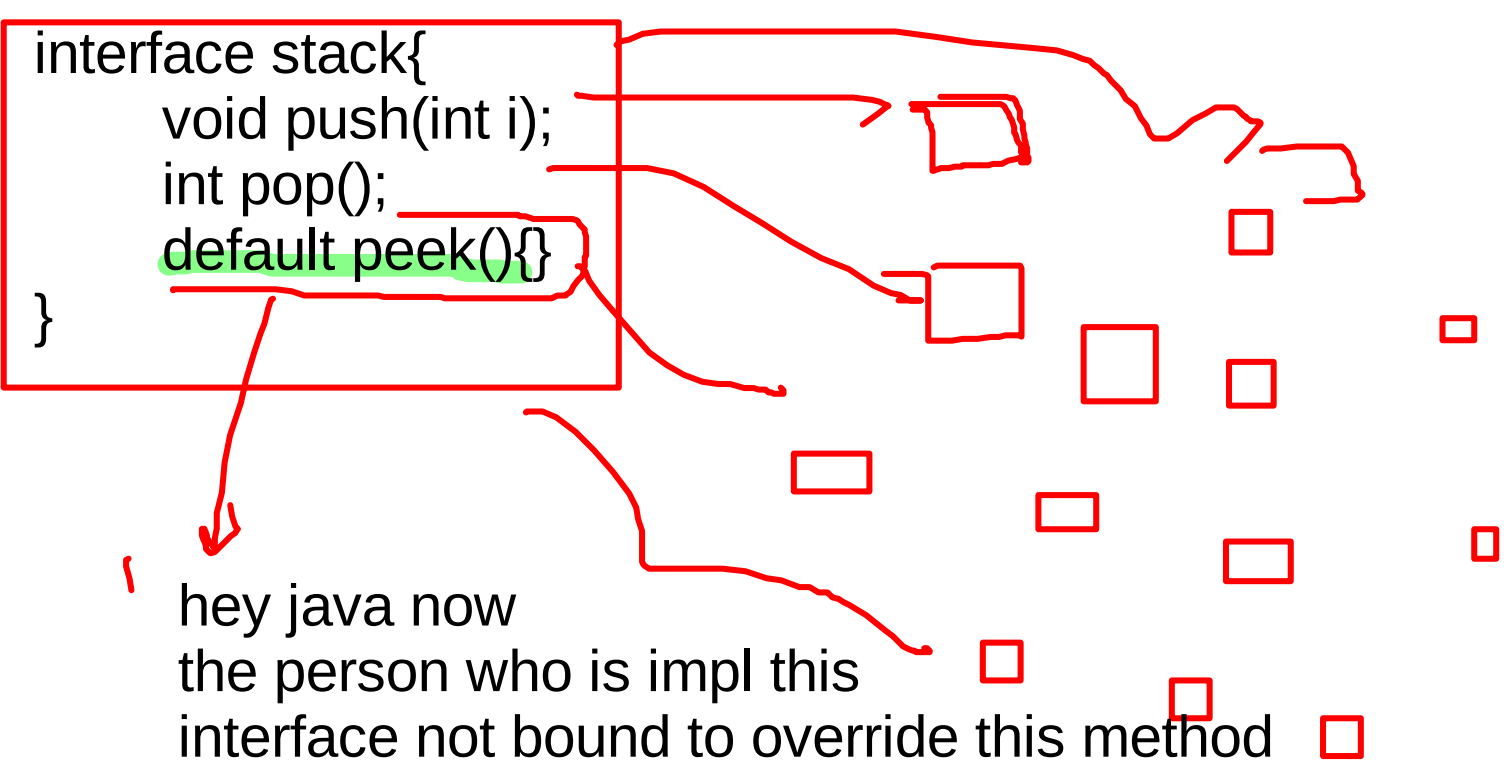
java

collection: aka readymade ds in java

java 8: stream procssing

SQL

```
interface stack{
        void push(int i);
        int pop();
        default peek(){}
}
```



hey java now
the person who is impl this
interface not bound to override this method
if i he want he can do that

:)

java 8

```
interface stack{
        void push(int i);
        int pop();
         static void fooStack(){}
        default peek(){}
}
```

java 9

```
interface stack{
        private void foo(){}
        void push(int i);
        int pop();
        public static void fooStack(){}
        default peek(){}
}
```

```
class Outer{
    private int i=33;
    class Inner{
        private int i=333;
        public void printI(){
            System.out.println(this.i);
            System.out.println(Outer.this.i);
        }
    }
    public void demo(this){
        Inner inner=this.new Inner();
        inner.printI();
    }
}
public class A_DemoInnerClass {
    public static void main(String[] args) {
        Outer outer=new Outer();
        outer.demo();
    }
}
```

Weak entity

```
class LinkedList {
    Node head; // head of list

    /* Linked list Node*/
    class Node {
        int data;
        Node next;

        // Constructor to create a new node
        // Next is by default initialized
        // as null
        Node(int d) { data = d; }
    }
}
```

```
    class University{


            class Dept{


            }
    }
```

Top level inner class

- **Non static inner class object can't be created without outer class instance**
- **All the private data of outer class is available to inner class**
- **Non static inner class can't have static members**

```
class A
{
    private int i=90;

    class B
    {
        int i=90;//
        void foo()
        {
            System.out.println("instance value of outer class:"+ A.this.i);

            System.out.println("instance value of inner class:"+this.i);
        }
    }
}
public class Inner1 {

    public static void main(String[] args) {

        A.B objectB=new A().new B();

        objectB.foo();
```

rgupta.mtech@gmail.com

Method local inner class

- Class define inside an method
- Can not access local data define inside method
- Declare local data final to access it

```
class A
{
    private int i=90;
    void foo()
    {
        int i=22;
        final int j=44;
        class B
        {
            void foofy()
            {
                //System.out.println("value of method local variable cant be access:"+i);
                System.out.println("value of method local" +
                        " can be accessed if it is declared final:"+ j);
            }
        }

        B b=new B();
        b.foofy();
    }

}
```

rgupta.mtech@gmail.com

Anonymous Inner class

- A way to implement polymorphism
- “On the fly”

```
interface Cookable
{
    public void cook();
}

class Food
{
    Cookable c=new Cookable() {
        @Override
        public void cook() {
            System.out.println("Cooked......");
        }
    };
}
```

rgupta.mtech@gmail.com

robust

try
catch
throw
thows
finally

Session 5:
Java Exception Handling, IO, Serilization

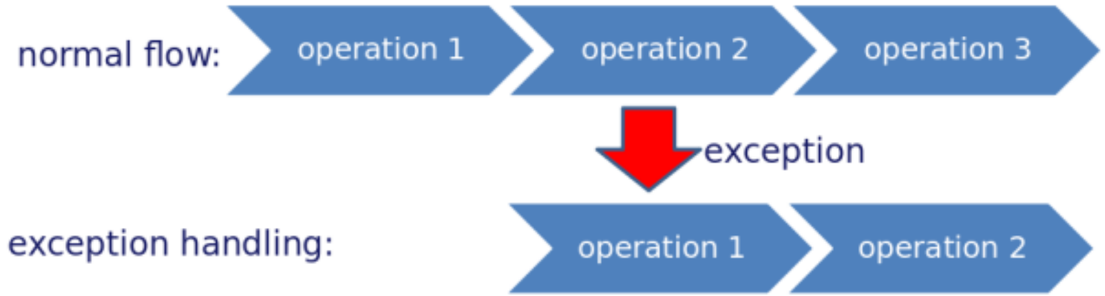
rgupta.mtech@gmail.com

```
try{
    Scanner sc = new Scanner(System.in);
    int a, b,c ;
    System.out.println("PE first no");
    a=sc.nextInt();
    System.out.println("PE sec no");
    b=sc.nextInt();

    c=a/b;
    System.out.println("value of c is : "+ c);
}catch (ArithmeticException ex){
    System.out.println("pls dont put demo as zero");
}catch (InputMismatchException ex){
    System.out.println("pls only enter ints");
}

 System.out.println("done");
```

What is Exception?

- **Exception** – is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- **Exception handling** is convenient way to handle errors

- Attempt to divide an integer by zero causes an exception to be thrown at run time.
- Attempt to call a method using a reference that is null. Attempting to open a nonexistent file for reading.
- JVM running out of memory.



- Exception handling do not correct abnormal condition rather it make our program robust i.e. make us enable to take remedial action when exception occurs...Help in recovering...

rgupta.mtech@gmail.com

Type of exceptions

1. Unchecked Exception Also called Runtime Exceptions

2. Checked Exception
Also called Compile time Exceptions

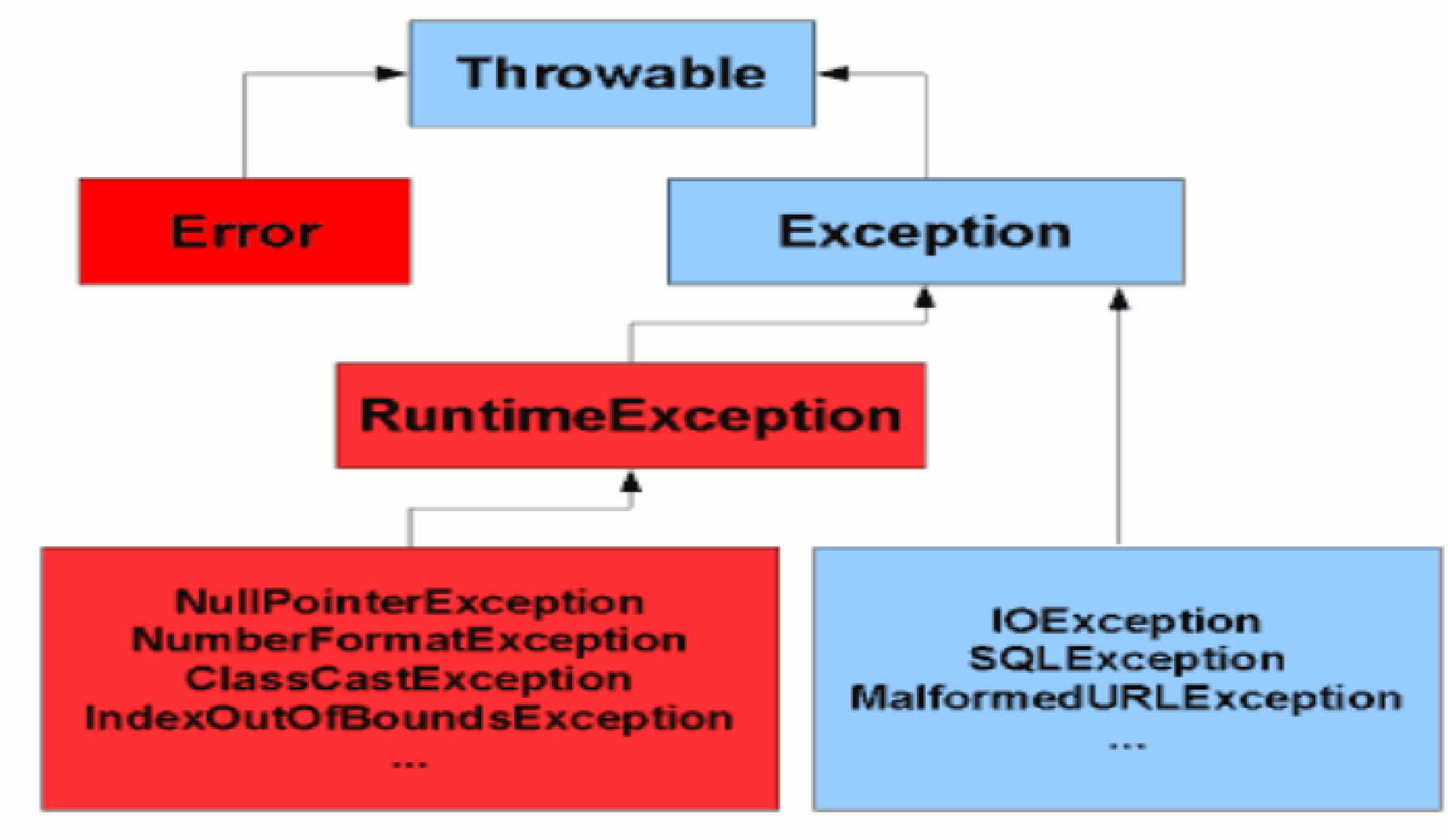
3. Error
Should not to be handled by programmer..like JVM crash

1. try
2. catch
3. throw
4. throws
5. finally

All exceptions happens at run time in programming and also in real life.....
for checked ex, we need to tell java we know about those problems for example
readLine() throws IOException

rgupta.mtech@gmail.com

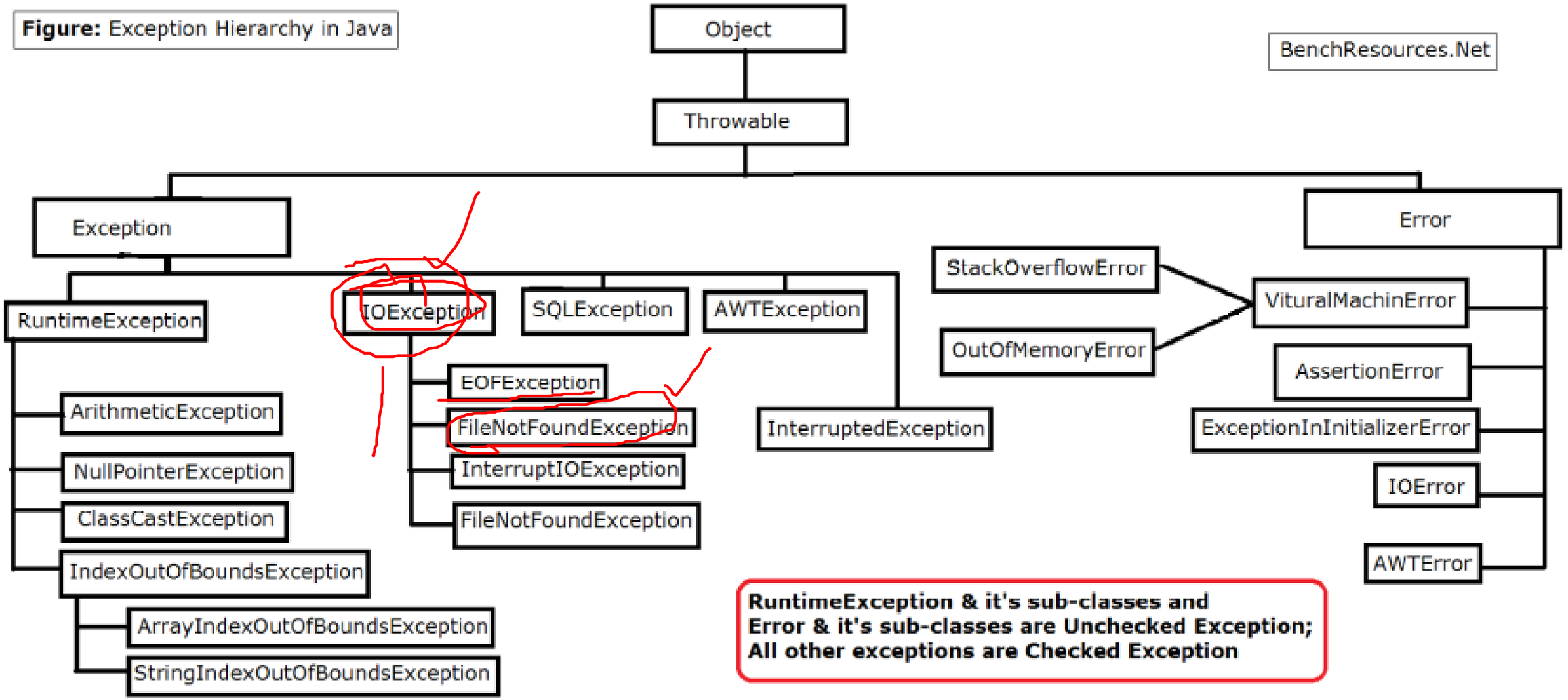
Exception Hierarchy



rgupta.mtech@gmail.com

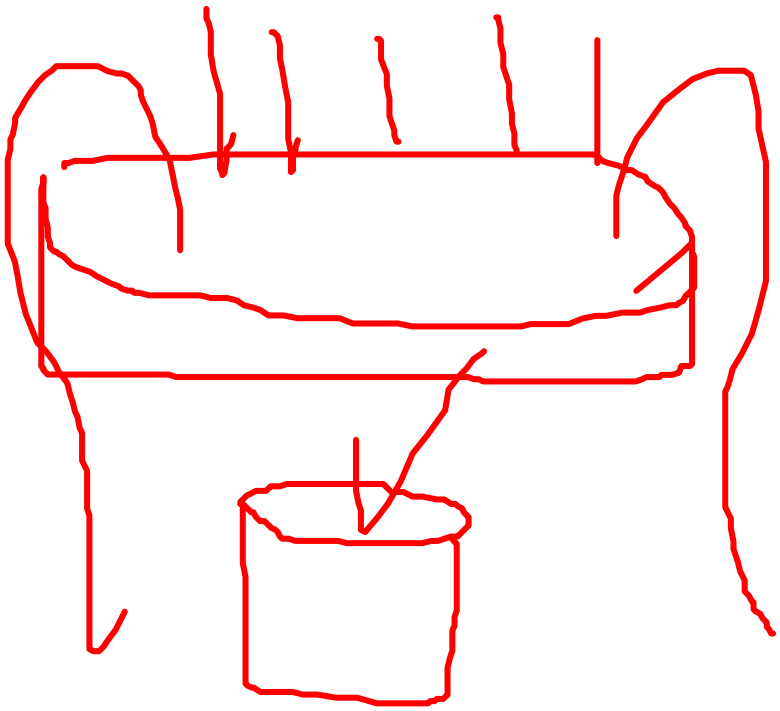
Exception Hierarchy

Figure: Exception Hierarchy in Java



rgupta.mtech@gmail.com

```
catch (Exception ex){
            System.out.println("handled ex");
        }
        catch (ArithmeticException ex){
            System.out.println("pls dont put demo as zero");
        }catch (InputMismatchException ex){
```



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

logic java

```
class Circle{
    private int radius;
    public Circle(int radius) throws InvalidRadiusException{
        this.radius = radius;
        if(radius<=0)
            throw new InvalidRadiusException("invalid radius "+ radius);
    }
    public double getPerimeter(){
        return Math.PI* radius*2;
    }
}
public class DemoBankAcc {
    public static void main(String[] args) {
       try{
           Circle c=new Circle(2);
           System.out.println(c.getPerimeter());
       }catch (InvalidRadiusException e){
           System.out.println(e.getMessage());
       }
    }
}
```

controller

```
class Window implements AutoCloseable {
    public Window(){
        System.out.println("ctr of window");
    }
    @Override
    public void close() throws IOException {
        System.out.println("close method");
    }
    public void doWork(){
        System.out.println("do work of window class");
    }
}
public class C_UnderstandingARM {
    public static void main(String[] args) {
        try(Window window=new Window()){
            window.doWork();
        }catch (Exception ex){
            System.out.println("some ex happens "+ ex);
        }
    }
}
```

Common Java Exceptions

- **ArithmeticException**
- **ArrayIndexOutOfBoundException**
- **ArrayStoreException**
- **FileNotFoundException**
- **IOException – general I/O failure**
- **NullPointerException – referencing a null object**
- **OutOfMemoryError**
- **SecurityException – when applet tries to perform an action not allowed by the browser's security setting.**
- **StackOverflowError**
- **StringIndexOutOfBoundException**

rgupta.mtech@gmail.com

finally

- **A finally clause is executed even if a return statement is executed in the try or catch clauses.**
- **An uncaught or nested exception still exits via the finally clause.**
- **Typical usage is to free system resources before returning, even after throwing an exception (close files, network links)**

```
try {
    // Protect one or more statements here
}
catch(Exception e) {
    // Report from the exception here
}
finally {
    // Perform actions here whether
    // or not an exception is thrown
}
```

rgupta.mtech@gmail.com

Java 7 Resource Management

Java 7 has introduced a new interface java.lang.AutoCloseable which is extended by java.io.Closeable interface. To use any resource in try-with-resources, it must implement AutoCloseable interface else java compiler will throw compilation error.

```
public class MyResource implements AutoCloseable{
    @Override
    public void close() throws Exception {
        System.out.println("Closing");
}    }
```

```
try (MyResource sr =
            new MyResource()) {
    //doSomething with sr
}
```

```
try (MyResource sr =
            new MyResource()) {
    //doSomething with sr
}
```

rgupta.mtech@gmail.com

Java 7 Resource Management

Instead of

```
public void oldTry() {
    try {
        fos = new
FileOutputStream("movies.txt");
        dos = new
DataOutputStream(fos);
        dos.writeUTF("Java 7 Block
Buster");
        } catch (IOException e) {
        e.printStackTrace();
        } finally {
        try {
            fos.close();
            dos.close();
            } catch (IOException e) {
            // log the exception
        }
    }
}
```

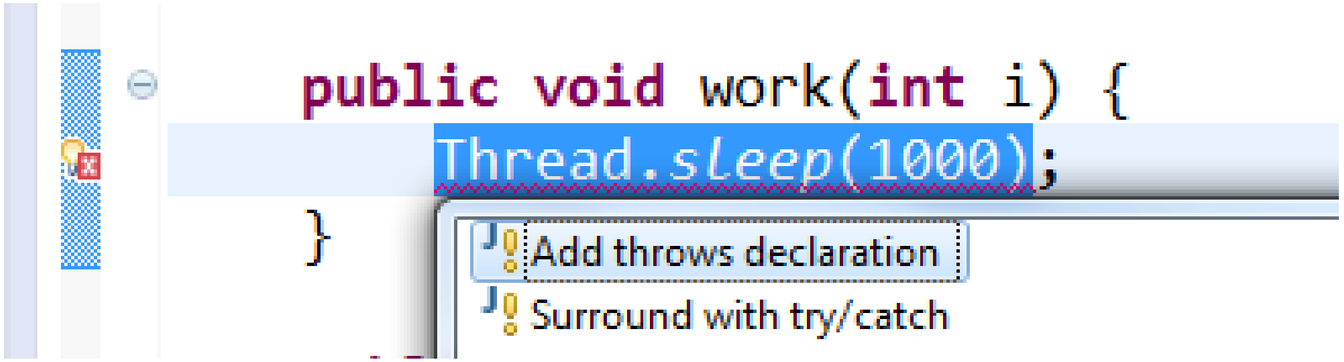
You can now say:

```
public void newTry() {
    try (FileOutputStream fos = new
FileOutputStream("movies.txt");
    DataOutputStream dos = new
DataOutputStream(fos)) {
        dos.writeUTF("Java 7 Block
Buster");
        } catch (IOException e) {
        // log the exception
    }
}
```

rgupta.mtech@gmail.com

Statement throws

If a method can throw an exception, which he does not handle, it must specify this behavior so that the calling code could take care of this exception.
Also there is the design throws, which lists the types of exceptions.
Except Error, RuntimeException, and their subclasses.



For example

```
double safeSqrt(double x)
      throws ArithmeticException {
    if (x < 0.0)
      throw new ArithmeticException();
  return Math.sqrt(x);
}
```

rgupta.mtech@gmail.com

Statement throw

You can throw exception using the **throw** statement

```
try {
    MyClass myClass = new MyClass( );
    if (myClass == null) {
        throw new NullPointerException("Messages");
    }
} catch (NullPointerException e) {
    // TODO
    e.printStackTrace( );
    System.out.println(e.getMessage( ));
}
```

rgupta.mtech@gmail.com

Summary: Dealing with Checked Exceptions

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

    FileReader fr = new FileReader("text.txt");

}
```

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

    try {
        FileReader fr = new FileReader("text.txt");
    } catch (FileNotFoundException e) {
        // Do something
    }

}
```

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

    FileReader fr = new FileReader("text.txt");

}
```

rgupta.mtech@gmail.com

Defining new exception

- **You can subclass RuntimeException to create new kinds of unchecked exceptions.**
- **Or subclass Exception for new kinds of checked exceptions.**
- **Why? To improve error reporting in your program.**

Create ***checked*** exception – MyException

```
// Creation subclass with two constructors
class MyException extends Exception {

  // Classic constructor with a message of
error
  public MyException(String msg) {
    super(msg);
  }

  // Empty constructor
  public MyException() {  }
}
```

rgupta.mtech@gmail.com

Defining new exception

If you create your own exception class from ***RuntimeException***, it's not necessary to write exception specification in the procedure.

```
class MyException extends RuntimeException { }

public class ExampleException {
    static void doSomthing(int n) {
        throw new MyException( );
    }
    public static void main(String[ ] args) {
        DoSomthing(-1);  // try / catch do not use
    }
}
```

rgupta.mtech@gmail.com

Stack Trace

The exception keeps being passed out to the next enclosing block until:

- **a suitable handler is found;**
- **or there are no blocks left to try and the program terminates with a stack trace**

If no handler is called, then the system prints a stack trace as the program terminates

- **it is a list of the called methods that are waiting to return when the exception occurred**
- **very useful for debugging/testing**

The stack trace can be printed by calling printStackTrace()

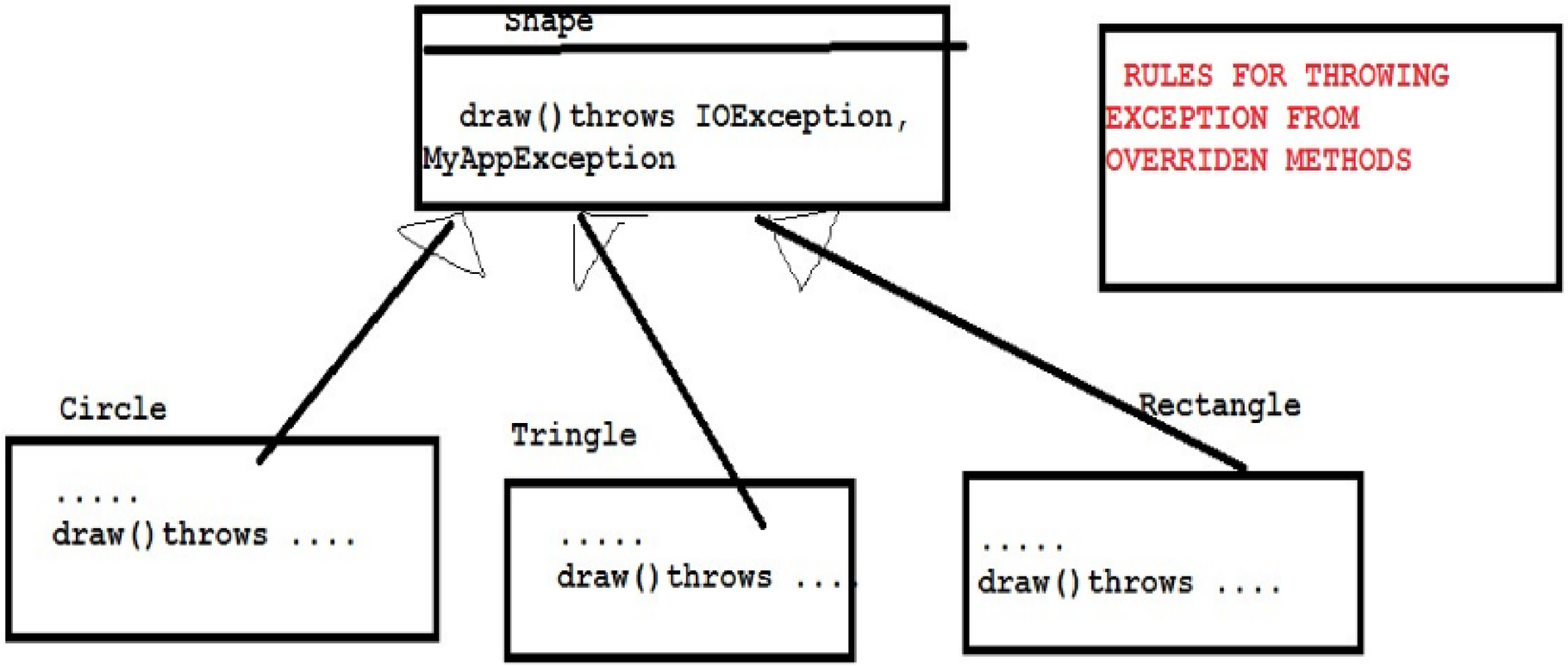
rgupta.mtech@gmail.com

Using a Stack Trace

```
// The getMessage and printStackTrace methods
public static void main( String[] args) {
    try {
        method1();
    } catch (Exception e) {
        System.err.println(e.getMessage() + "\
n");
        e.printStackTrace();
    }
}
```

rgupta.mtech@gmail.com

Rules for throwing exception from overridden methods



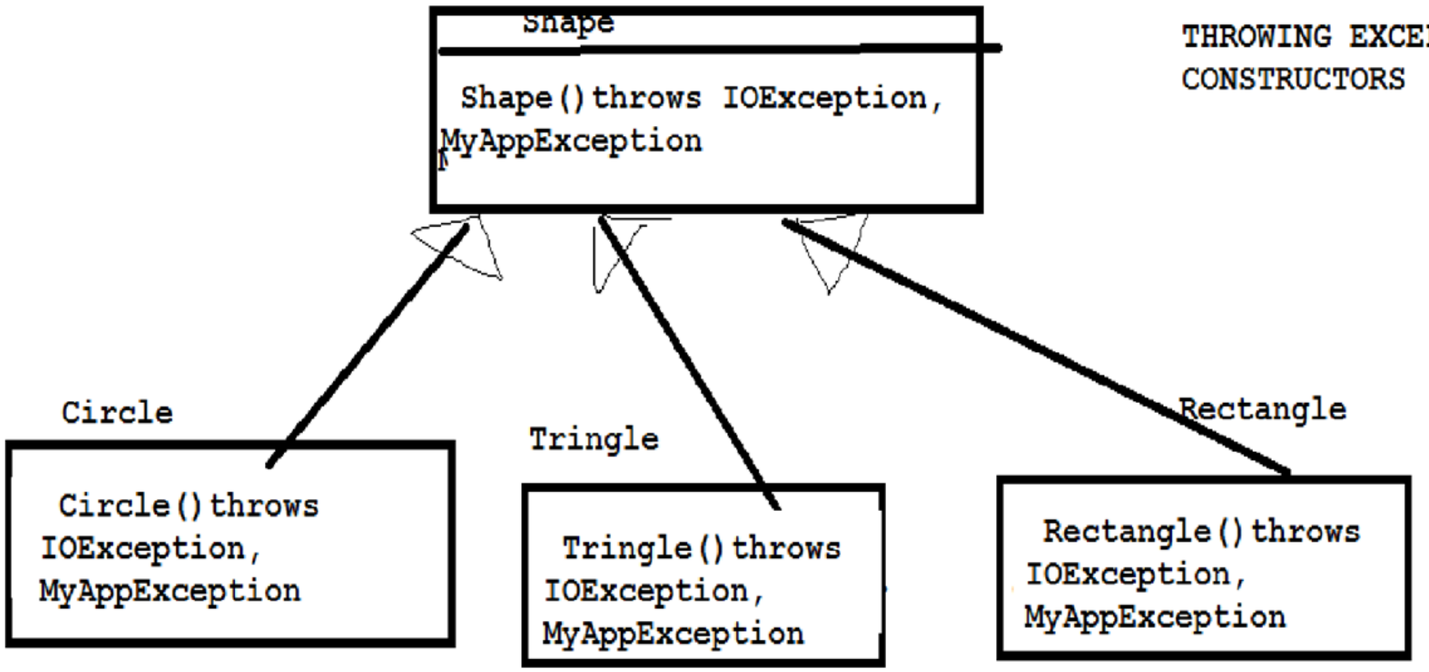
* Any checked exception throws here must be subclass of IOExcetion and MyAppException

* can throws any unchecked exceptions

*Not throwing any exception is also valid

rgupta.mtech@gmail.com

Rules for throwing exception from constructors



THROWING EXCEPTION FROM CONSTRUCTORS

* Must throws IOException and MyAppException

* May throws additional checked exceptions

rgupta.mtech@gmail.com

Rules for throwing exception from overloaded and other methods

Throwing Exception from Overloaded and other methods...

```
class MyClass{

    ...
    void MyMethod(...)throws ...{}

    void MyMethod(...,...)throws ....{}

    void MyOtherMethod(...)throws ....{}
}
```

*Not an overriden method
*also not a const

⇩

hence can throw any exception irrespective of exception thrown by other overloaded methods

Can throw any exception...

rgupta.mtech@gmail.com

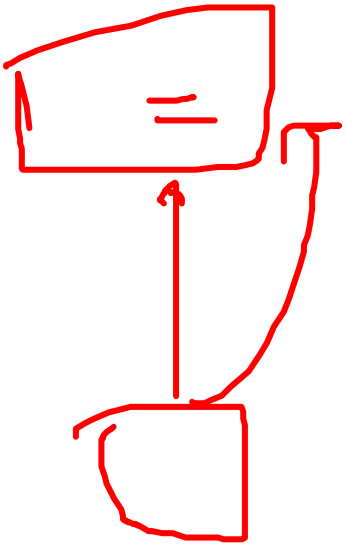
Suppressed Exceptions

```
public static void main(String[] args) {
    try{
        foo();
    }catch (Exception e){
        System.out.println(e);
       Throwable[]throwables= e.getSuppressed();
       for(Throwable t: throwables){
           System.out.println(t);
       }
    }
}
```

```
private static void foo()throws Exception {
    Exception t=null;
    try{
        throw  new TryEx();
    }catch (Exception ex){
        t=ex;
    }
    finally {
       FinallyEx e=new FinallyEx();
       if(t!=null){
           e.addSuppressed(t);
       }
       throw e;
    }
}
```

rgupta.mtech@gmail.com

```
class A{
    private int a;
    A(int a){
      this.a=a;
    }
}
class B extends A{
    private int b;
    B(int a, int b){
        super(a);
        this.b=a;
    }
}
public class DemoDoubt {
    public static void main(String[] args) {
        A a=new B(1,3);

    }
}
```



Suppressed Exceptions with ARM

```
class Door implements  AutoCloseable{

    public void doLogicDoor(){
        System.out.println("logic of door");
        throw new NullPointerException("NPE");
    }
    public Door() {
        System.out.println("ctr of door");
    }
    @Override
    public void close() throws Exception {
        System.out.println("closing the door");
        throw new ArithmeticException("ArithmeticException");
    }
}
```

```
    try(Door door=new Door()){
        door.doLogicDoor();
    }catch (Exception ex){
        System.out.println(ex.getClass().getName());
         Throwable[] throwables= ex.getSuppressed();
         Arrays.asList(throwables).forEach(e-> System.out.println(e));
    }
```

rgupta.mtech@gmail.com

exception handling best practices

rgupta.mtech@gmail.com

Exception Handling Best Practices

- Use Specific Exceptions – we should always *throw* and *catch* specific exception classes so that caller will know the root cause of exception easily and process them. This makes debugging easy and helps client application to handle exceptions appropriately.
- Throw early - we should try to throw exception as early as possible.
- Catch late – we should catch exception only when we can handle it appropriate.
- Close resources - we should close all the resources in finally block or use Java 7 block try-with-resources.
- Do Not Use Exceptions to Control Application Flow

rgupta.mtech@gmail.com

exception handling best practices tips #1

- **Handle Exceptions close to its origin**
  - Does NOT mean "catch and swallow" (i.e. suppress or ignore exceptions)

Example

```
try {
 // code that is capable of throwing a XyzException
} catch (XyzException e) {
 // do nothing or simply log and proceed
}
```

  - It means, "log and handle the exception right there" or "log and throw the exception up the method call stack using a custom exception relevant to that source layer" and let it be handled later by a method up the call stack.
    - DAO layer – DataAccessException
    - Business layer – Application's Custom Exception (example - SKUException)

The approach "log and handle the exception right there" makes way to use the specific exception type to differentiate exceptions and handle exceptions in some explicit manner.

The approach "log and throw the exception up the method call stack using a custom exception relevant to that source layer" – makes way for creation of groups of exceptions and handling exceptions in a generic manner.

rgupta.mtech@gmail.com

exception handling best practices tips #1 continue...

Note

When catching an exception and throwing it using an exception relevant to that source layer, make sure to use the construct that passes the original exception's cause. This will help preserve the original root cause of the exception.

```
try {
    // code that is capable of throwing a SQLException
} catch (SQLException e) {

    // log technical SQL Error messages, but do not pass
    // it to the client. Use user-friendly message instead

    LOGGER.error("An error occurred when searching for the SKU
    details" + e.getMessage());

    throw new DataAccessException("An error occurred when searching
    for the SKU details", e);
}
```

exception handling best practices tips #2

- **Log Exceptions just once and log it close to its origin**

  Logging the same exception stack trace more than once can confuse the programmer examining the stack trace about the original source of exception. So, log Exceptions just once and log it close to its origin.

```
try {

    // Code that is capable of throwing a XyzException

} catch (XyzException e) {

    // Log the exception specific information.
    // Throw exception relevant to that source layer

}
```

rgupta.mtech@gmail.com

exception handling best practices tips #2 continue

Note

There is an exception to this rule, in case of existing code that may not have logged the exception details at its origin.

In such cases, it would be required to log the exception details in the first method up the call stack that handles that exception. But care should be taken NOT to COMPLETELY overwrite the original exception's message with some other message when logging.

Example

DAO Layer:

```
try {
    // code that is capable of throwing a SQLException
} catch (SQLException e) {
    // Note that LOGGING has been missed here
    throw new DataAccessException("An error occurred when
    processing the query.", e);
}
```

rgupta.mtech@gmail.com

exception handling best practices tips #2 continue

Since logging was missed in the exception handler of the DAO layer, it is mandated in the exception handler of the next enclosing layer – in this example it is the (Business/Processor) layer.

Business/Processor Layer:

```
try {
      // code that is capable of throwing a DataAccessException

} catch (DataAccessException e) {
     // logging is mandated here as it was not logged
     // at its source (DAO layer method)
     LOGGER.error(e.getMessage());
     throw new SKUException(e.getMessage(), e);
 }
```

rgupta.mtech@gmail.com

exception handling best practices tips #3

- **Do not catch "Exception"**

Accidentally swallowing RuntimeException

```
try {
    doSomething();
} catch (Exception e) {
    LOGGER.error(e.getMessage());
}
```

This code

- also captures any RuntimeExceptions that might have been thrown by doSomething,
- ignores unchecked exceptions and
- prevents them from being propagated.

So, all checked exceptions should be caught and handled using appropriate catch handlers. And the exceptions should be logged and thrown to the outermost layer (i.e. the method at the top of the calling stack) using application specific custom exceptions relevant to that source layer.

rgupta.mtech@gmail.com

exception handling best practices tips #4

- **Handle Exceptions before sending response to Client**

  The layer of code component (i.e. the method at the top of the calling stack) that sends back response to the client, has to do the following:

  - catch ALL checked exceptions and handle them by creating proper error response and send it back to client.
  - NOT allow any checked exception to be "thrown" to the client.
  - handle the Business layer exception and all other checked exceptions raised from within the code in that layer separately.

Examples of such components are:

- Service layer Classes in Web Service based applications
- Action Classes in Struts framework based applications

rgupta.mtech@gmail.com

exception handling best practices tips #4 continue

- **Handle Exceptions before sending response to Client**

  The layer of code component (i.e. the method at the top of the calling stack) that sends back response to the client, has to do the following:

  - catch ALL checked exceptions and handle them by creating proper error response and send it back to client.
  - NOT allow any checked exception to be "thrown" to the client.
  - handle the Business layer exception and all other checked exceptions raised from within the code in that layer separately.

Examples of such components are:

- Service layer Classes in Web Service based applications
- Action Classes in Struts framework based applications

rgupta.mtech@gmail.com

exception handling best practices tips #4 continue

An exception to handling ‘Exception’ – Case 1

There would be situations (although rarely) where the users would prefer a user-friendly/easy to understand message to be shown to them, instead of the system defined messages thrown by unrecoverable exceptions.

In such cases, the method at the top of the calling stack, which is part of the code that sends response to the client is expected to handle all unchecked exceptions thrown from within the ‘try’ block.

By doing this, technical exception messages can be replaced with generic messages that the user can understand. So, a catch handler for ‘Exception’ can be placed in it.

This is an exception to best practice #3 and is only for the outermost layer. In other layers downstream in the layered architecture, catching ‘Exception’ is not recommended for reasons explained under best practice #3.

rgupta.mtech@gmail.com

exception handling best practices tips #4 continue

Example

```
try {
      // Code that is capable of throwing a SKUException
      // (a custom exception in this example application)
} catch (SKUException e) {
      // Form error response using the exception's data,
      // error code and/or error message

} catch (Exception e) {

      // Log the exception related message here, since this block is
      // expected to get only the unchecked exceptions
      // that had not been captured and logged elsewhere in the code,
      // provided the exception handling and logging are properly
      // handled at the other layers in the layered architecture.

      // Form error response using the exception's data,
      // error code and/or error message
}
```

rgupta.mtech@gmail.com

exception handling best practices tips #4 continue

An exception to handling 'Exception' – Case 2

Certain other exceptional cases justify when it is handy and required to catch generic Exceptions. These cases are very specific but important to large, failure-tolerant systems.

Consider a request processing system that reads requests from a queue of requests and processes them in order.

```
public void processAllRequests() {
   Request req = null;
    try {
            while (true) {
                    req = getNextRequest();
                    if (req != null) {
                            processRequest(req); // throws BadRequestException
                    } else { // Request queue is empty, must be done
                            break;
                    }
            }
   } catch (BadRequestException e) {
            log.error("Invalid request:" + req, e);
   }
}
```

exception handling best practices tips #4 continue

An exception to handling 'Exception' – Case 2 (Contd..)

With the above code, if any exception occurs while the request is being processed (either a BadRequestException or *any* subclass of RuntimeException including NullPointerException), then that exception will be caught *outside* the processing 'while' loop.

So, any error causes the processing loop to stop and any remaining requests *will not* be processed. That represents a poor way of handling an error during request processing.

A better way to handle request processing is to make two significant changes to the logic.

1) Move the try/catch block inside the request-processing loop. That way, any errors are caught and handled inside the processing loop, and they do not cause the loop to break. Thus, the loop continues to process requests, even when a single request fails.

2) Change the try/catch block to catch a generic Exception, so *any* exception is caught inside the loop and requests continue to process.

exception handling best practices tips #4 continue

An exception to handling 'Exception' – Case 2 (Contd..)

```
public void processAllRequests() {
  while (true) {
        Request req = null;
        try {
              req = getNextRequest();
              if (req != null) {
                    processRequest(req); // throws BadRequestException
              } else { // Request queue is empty, must be done
                    break;
              }
        } catch (BadRequestException e) {
              log.error("Invalid request:" + req, e);
        }
  }
}
```

rgupta.mtech@gmail.com

exception handling best practices tips #4 continue

An exception to handling 'Exception' – Case 2 (Contd..)

Catching a generic Exception sounds like a direct violation of the maxim suggested in best practice #3 —and it is. But the circumstance discussed is a specific and special one. In this case, the generic Exception is being caught to prevent a single exception from stopping an entire system.

In situations where requests, transactions or events are being processed in a loop, that loop needs to continue to process even when exceptions are thrown during processing.

rgupta.mtech@gmail.com

exception handling best practices tips #5

- **Handling common Runtime Exceptions**
  - NullPointerException
    - It is the developer's responsibility to ensure that no code can throw it.
    - Run CodePro and add null reference checks wherever it has been missed.
  - NumberFormatException, ParseException

    Catch these and create new exceptions specific to the layer from which it is thrown (usually from business layer) using user-friendly and non technical messages.
  - To avoid ClassCastException, check the type of the class to be cast using the instanceof operator before casting.
  - To avoid IndexOutOfBoundsException, check the length of the array before trying to work with an element of it.
  - To avoid ArithmeticException, make sure that the divisor is not zero before computing the division.

rgupta.mtech@gmail.com

exception handling best practices tips #5 continue

- **Example**

```
try {

   int item = Integer.parseInt(itemNumber);

} catch (NumberFormatException nfe) {

   LOGGER.error("SKU number is invalid and not a number");

   throw new SKUException("SKU number is invalid and not a number",
   nfe);

}
```

- All other unchecked exceptions (RuntimeExceptions) will be caught and handled by the 'Exception' handler in the outermost layer (as explained in Best Practice #4 - Case 1).

rgupta.mtech@gmail.com

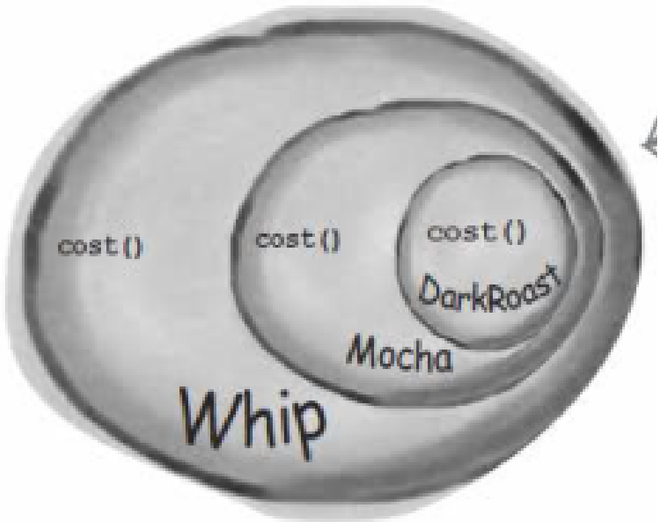
exception handling best practices tips #6

- **Document Exceptions Thrown in Javadoc**

  For each method that throws checked exceptions, document each exception thrown with a @throws tag in its Javadoc, including the condition under which the exception is thrown.

rgupta.mtech@gmail.com

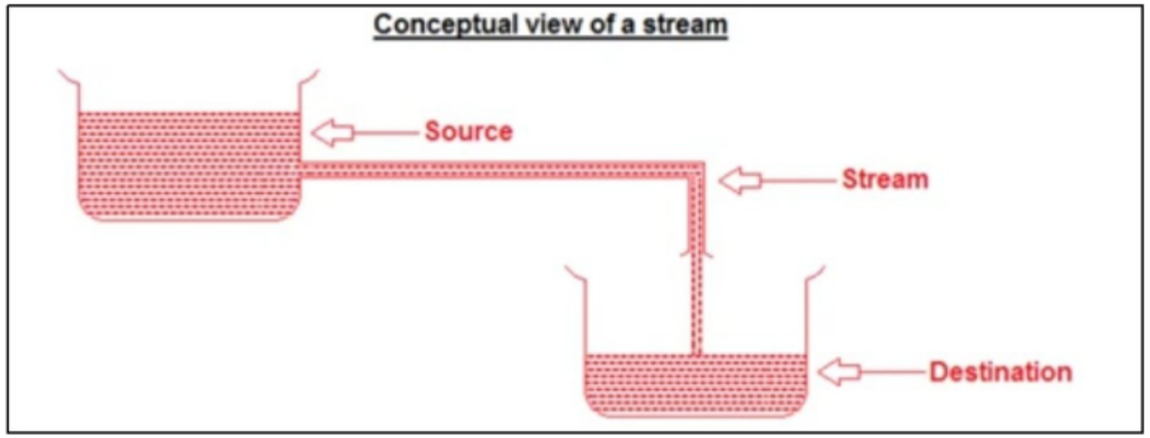
Session 6:
Java IO, Serilization



rgupta.mtech@gmail.com

Stream and IO

- Stream is logical connection bw source and destination
- Stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.
- **2 types of streams:**
  - byte : for binray data, contain 0,1 sequence
  - All byte stream class are like XXXXXXXXStream character: for **text data**
  - All char stream class are like XXXXXXXXReader/ XXXXXXWriter



rgupta.mtech@gmail.com

Stream I/O in Standard I/O (java.io Package)

A stream is a sequential and contiguous one-way flow of data (just like water or oil flows through the pipe). It is important to mention that Java does not differentiate between the various types of data sources or sinks (e.g., file or network) in stream I/O. They are all treated as a sequential flow of data. Input and output streams can be established from/to any data source/sink, such as files, network, keyboard/console or another program

Stream I/O operations involve three steps:

1. Open an input/output stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate I/O stream instance.
2. Read from the opened input stream until "end-of-stream" encountered, or write to the opened output stream (and optionally flush the buffered output).
3. Close the input/output stream.

rgupta.mtech@gmail.com

java.io.File

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Constructs a File instance based on the given path string.

public File(URI uri)
// Constructs a File instance by converting from the given file-URI "file://...."
```

For examples,

```
File file = new File("in.txt");     // A file relative to the current working directory
File file = new File("d:\\myproject\\java\\Hello.java");  // A file with absolute path
File dir  = new File("c:\\temp");   // A directory
```

For applications that you intend to distribute as JAR files, you should use the URL class to reference the resources, as it can reference disk files as well as JAR'ed files , for example,

```
java.net.URL url = this.getClass().getResource("icon.png");
```

Verifying Properties of a File/Directory

```
public boolean exists()       // Tests if this file/directory exists.
public long length()          // Returns the length of this file.
public boolean isDirectory()  // Tests if this instance is a directory.
public boolean isFile()       // Tests if this instance is a file.
public boolean canRead()      // Tests if this file is readable.
public boolean canWrite()     // Tests if this file is writable.
public boolean delete()       // Deletes this file/directory.
public void deleteOnExit()    // Deletes this file/directory when the program terminates.
public boolean renameTo(File dest) // Renames this file.
public boolean mkdir()        // Makes (Creates) this directory.
```

rgupta.mtech@gmail.com

java.io.File

(Advanced) List Directory with Filter

You can apply a filter to `list()` and `listFiles()`, to list only files that meet a certain criteria.

```
public String[] list(FilenameFilter filter)
public File[] listFiles(FilenameFilter filter)
public File[] listFiles(FileFilter filter)
```

The `interface java.io.FilenameFilter` declares one `abstract` method:

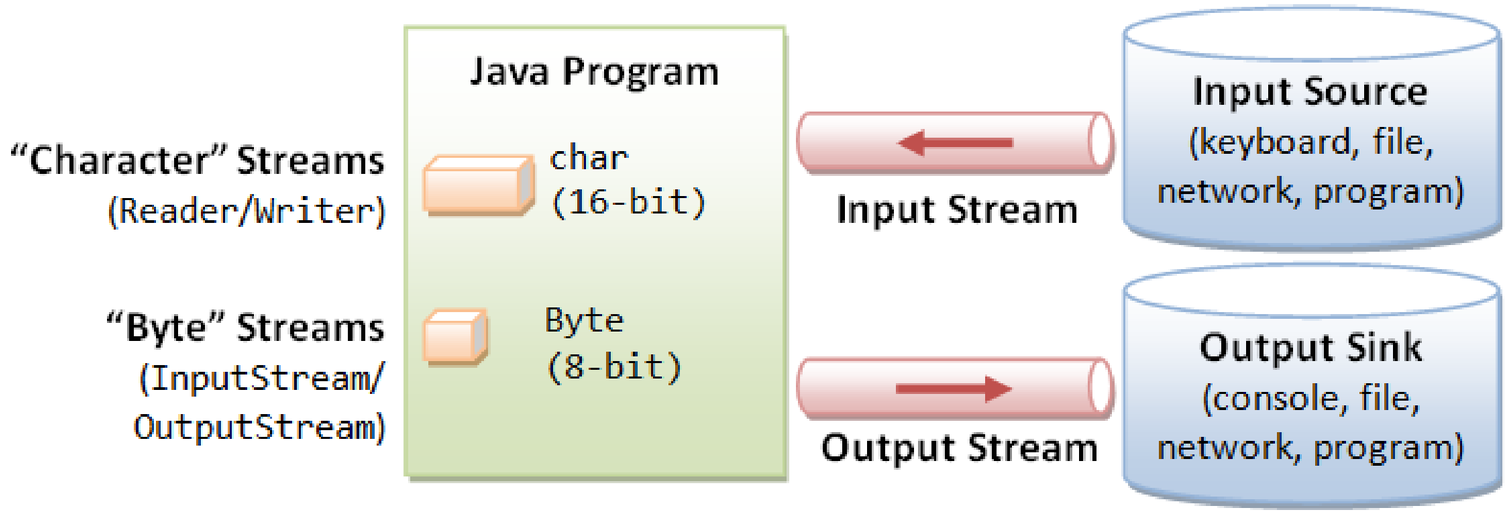
```
public boolean accept(File dirName, String fileName)
```

The `list()` and `listFiles()` methods does a *call-back* to `accept()` for each of the file/sub-directory produced. You can program your filtering criteria in `accept()`. Those files/sub-directories that result in a `false` return will be excluded.

Example: The following program lists only files that meet a certain filtering criteria.

```
// List files that end with ".java"
import java.io.File;
import java.io.FilenameFilter;
public class ListDirectoryWithFilter {
   public static void main(String[] args) {
      File dir = new File(".");   // current working directory
      if (dir.isDirectory()) {
         // List only files that meet the filtering criteria
         //  programmed in accept() method of FilenameFilter.
         String[] files = dir.list(new FilenameFilter() {
            public boolean accept(File dir, String file) {
               return file.endsWith(".java");
            }
         });  // an anonymous inner class as FilenameFilter
         for (String file : files) {
            System.out.println(file);
         }
      }
   }
```

Stream I/O in Standard I/O (java.io Package)



Internal Data Formats:

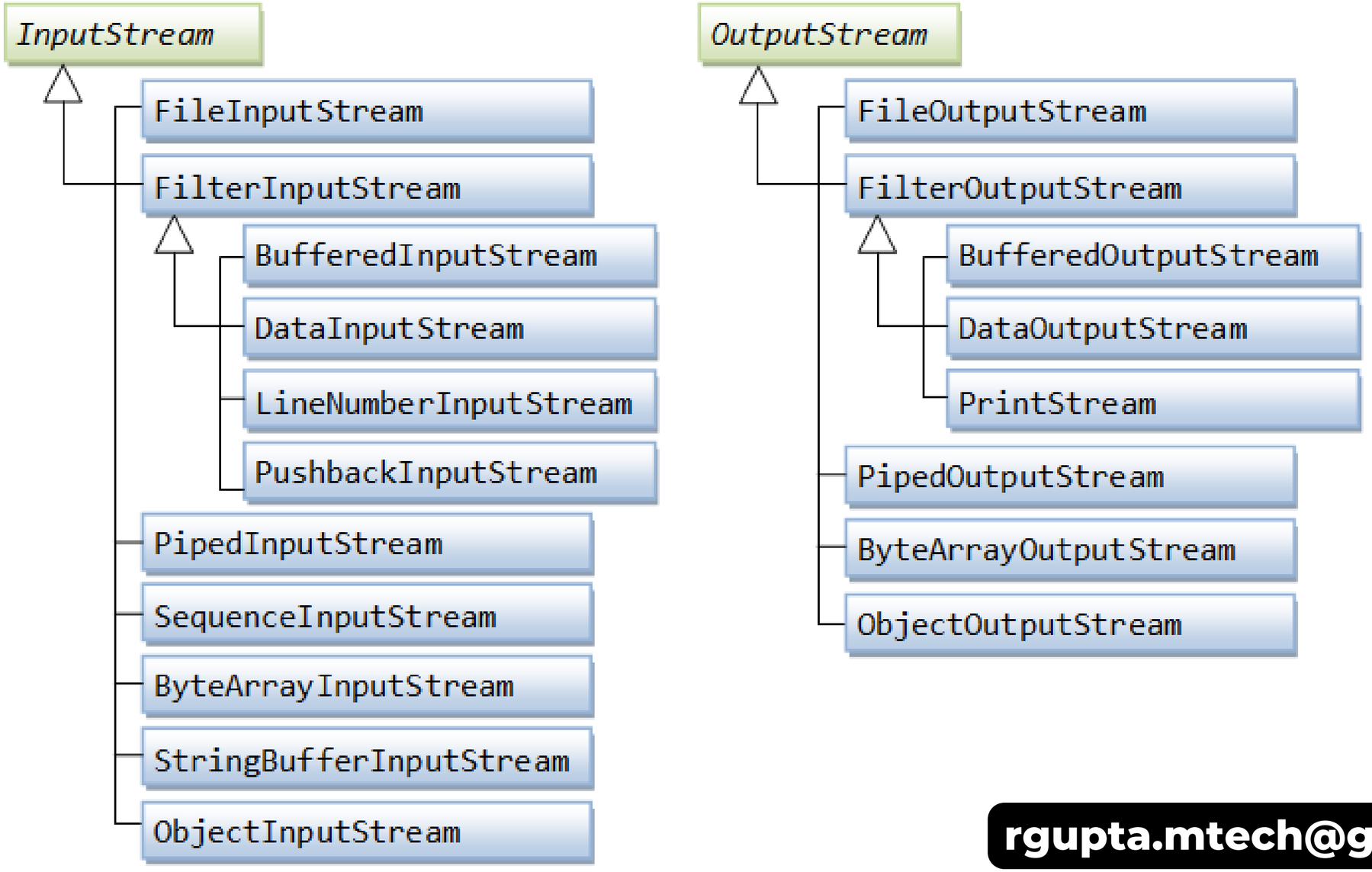
- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

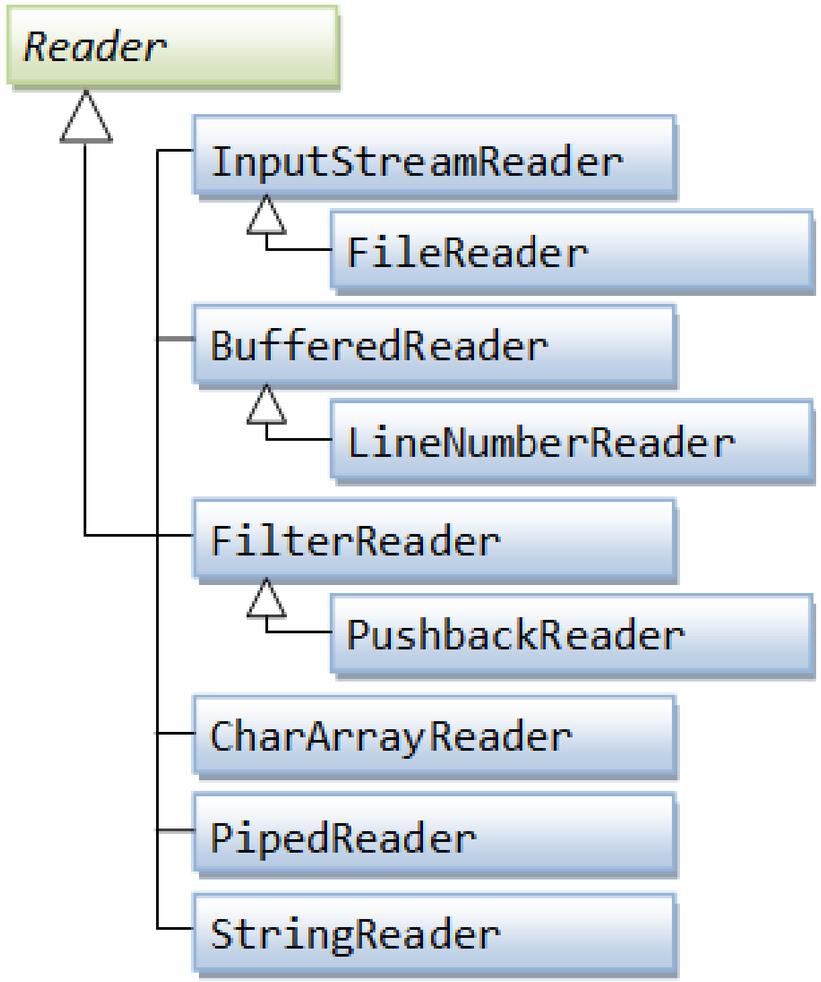
rgupta.mtech@gmail.com

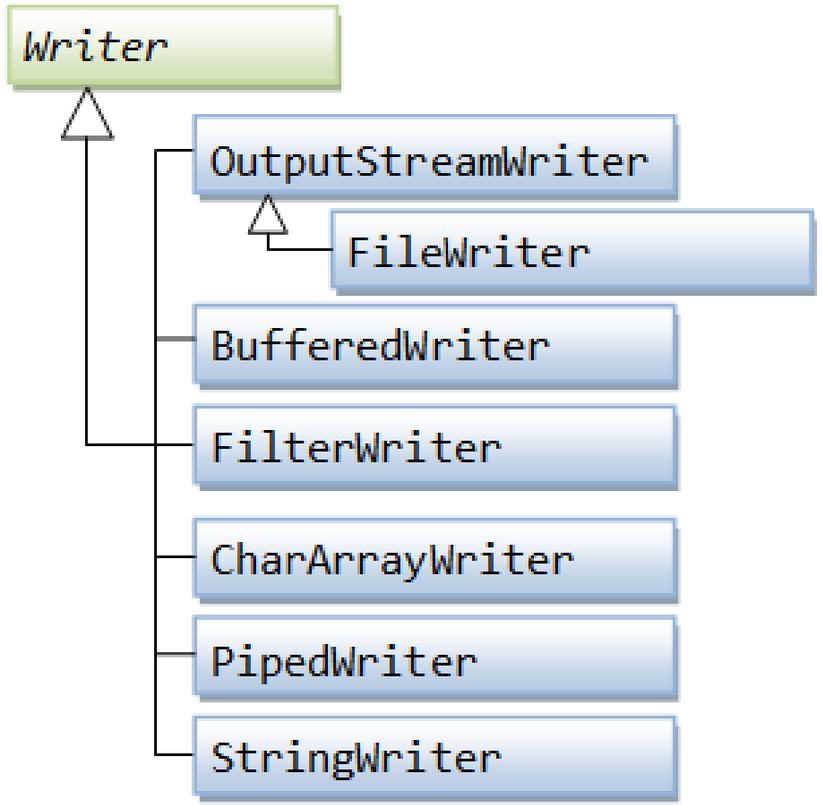
Byte-Based I/O & Byte Streams



rgupta.mtech@gmail.com

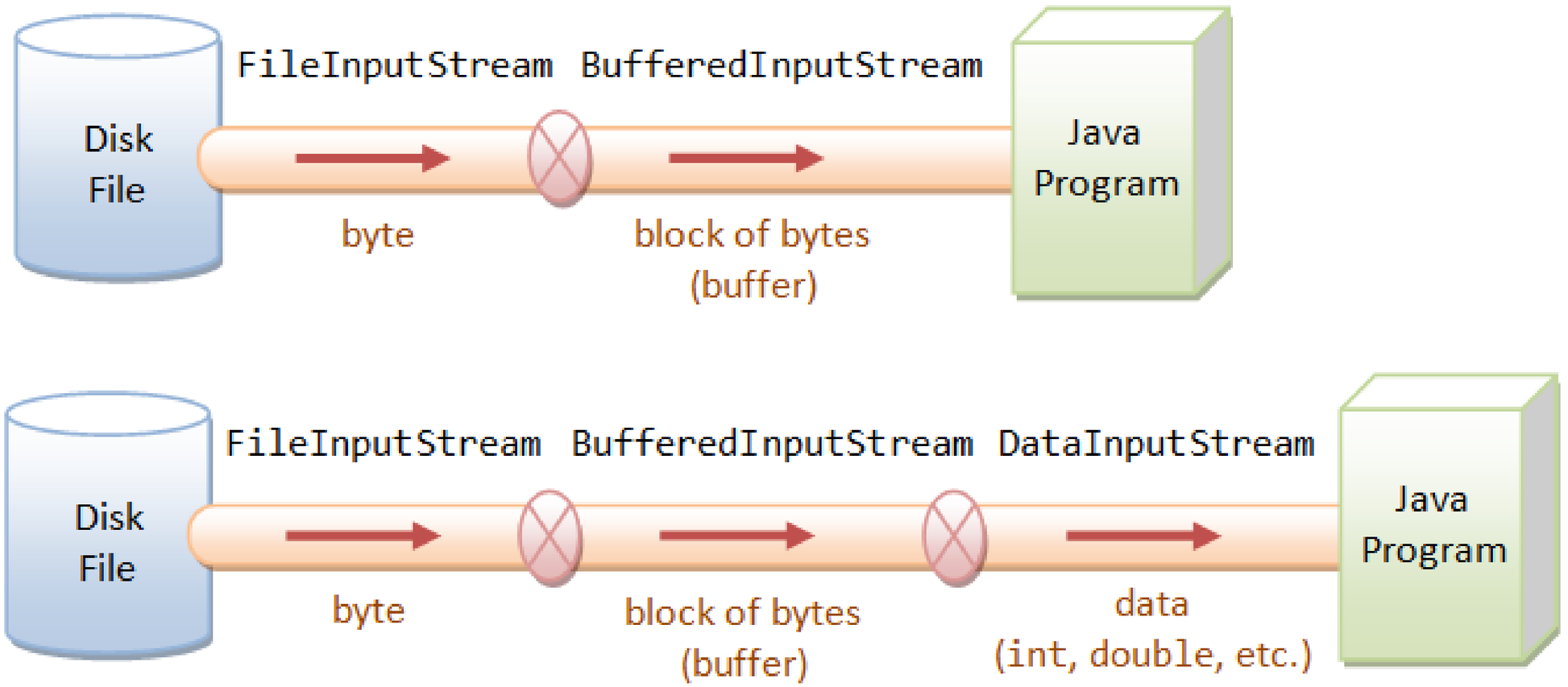
Character-Based I/O & Character Streams





rgupta.mtech@gmail.com

Layered (or Chained) I/O Streams



rgupta.mtech@gmail.com

File: using file, creating directories

File abstraction that represent file and directories

File f=new File("....");
boolean flag= file.createNewFile();
boolean flag=file.mkDir();
boolean flag=file.exists();

```
public class DemoFileWriter {

    public static void main(String[] args) {
        try {
            BufferedWriter bw=new BufferedWriter
                    (new FileWriter("demo5.txt"));

            bw.write("java is key");
            bw.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

}
```

rgupta.mtech@gmail.com

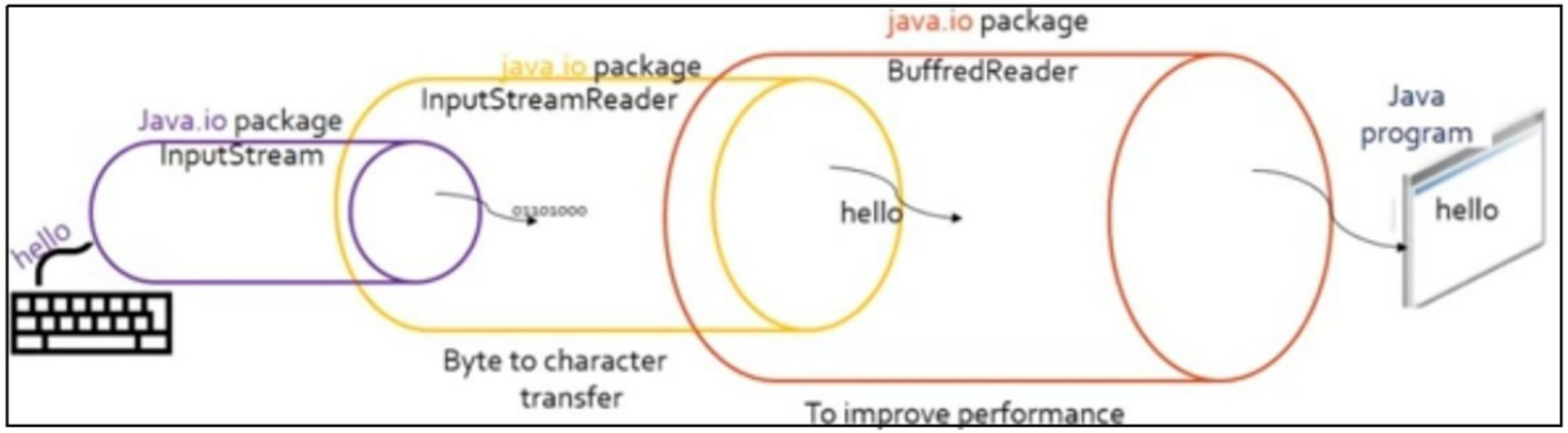
Buffered Reader and BufferedWriter

Reading form console

```
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
```

Reading form file

```
BufferedReader br=new BufferedReader(new FileReader(new
 File("c:\\raj\\foo.txt")));
```



rgupta.mtech@gmail.com

Buffered Reader and BufferedWriter

```
try(BufferedReader br = new BufferedReader(new FileReader(new File( pathname: "data.txt")))){
    String line=null;
    while ((line=br.readLine())!=null){
        String []tokens=line.split( regex: " ");
        for(String token: tokens){
            System.out.println(token);
        }
    }
}catch (FileNotFoundException ex){
    ex.printStackTrace();
}catch (IOException ex){
    ex.printStackTrace();
}catch (Exception ex){
    ex.printStackTrace();
}
```

rgupta.mtech@gmail.com

IO and Performance

Copying a photo without buffer

```
//time taken: 807 ms
FileInputStream fis=new FileInputStream( name: "/home/raj/Desktop/photo/mali.jpg");
FileOutputStream fos=new FileOutputStream( name: "/home/raj/Desktop/photo/mali_copy.jpg");

long start=System.currentTimeMillis();

int byteRead=1;
while ((byteRead=fis.read())!=-1){
    fos.write(byteRead);
}
fis.close();
fos.close();
long end=System.currentTimeMillis();
System.out.println("time taken: "+ (end-start)+" ms");
```

rgupta.mtech@gmail.com

IO and Performance

Copying a photo with our own buffer

```
//time taken: 807 ms
//time taken: 1 ms
FileInputStream fis=new FileInputStream( name: "/home/raj/Desktop/photo/mali.jpg");
FileOutputStream fos=new FileOutputStream( name: "/home/raj/Desktop/photo/mali_copy.jpg");

long start=System.nanoTime();

byte[] byteBuffer=new byte[4*1024];

int numberOfByteRead=1;
while ((numberOfByteRead=fis.read(byteBuffer))!=-1){
    fos.write(byteBuffer,  off: 0, numberOfByteRead);
}
fis.close();
fos.close();
long end=System.nanoTime();
System.out.println("time taken: "+ (end-start)+" ns");
```

rgupta.mtech@gmail.com

IO and Performance

Copying a photo with BufferedInputSteam

```
BufferedInputStream fis=new BufferedInputStream(new FileInputStream( name: "/home/raj/Desktop/photo/mali.jpg"));
BufferedOutputStream fos=new BufferedOutputStream(new FileOutputStream( name: "/home/raj/Desktop/photo/mali_copy.jpg"));

long start=System.currentTimeMillis();
 int byteRead=1;
 while ((byteRead=fis.read())!=-1){
     fos.write(byteRead);
 }
 fis.close();
 fos.close();
 long end=System.currentTimeMillis();
 System.out.println("time taken: "+ (end-start)+" ms");
```

rgupta.mtech@gmail.com

Using data output steam and data input stream

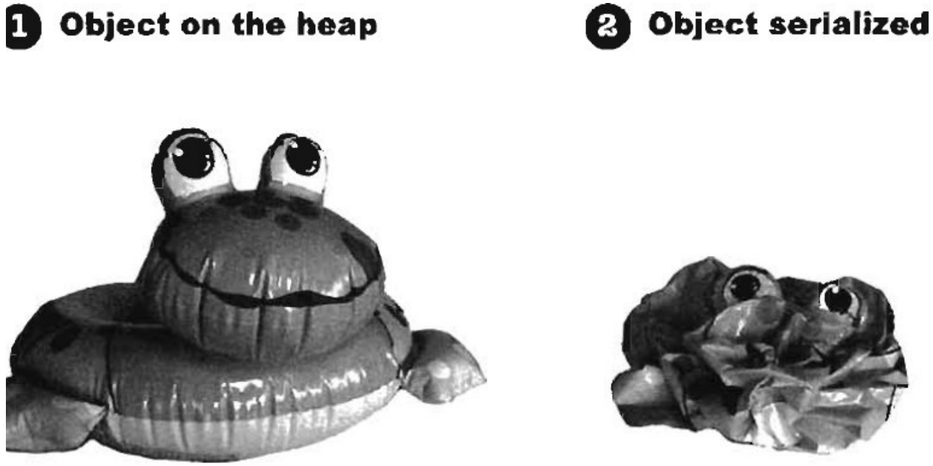
```
try {
    //writing in file
    DataOutputStream dos = new DataOutputStream
            (new FileOutputStream(
            new File("foo.txt")));
    for(int i=0; i<10;i++){
        dos.writeInt(i);
        dos.writeDouble(6.6);
        dos.writeShort(i);
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (Exception ex) {
    ex.printStackTrace();
}
```

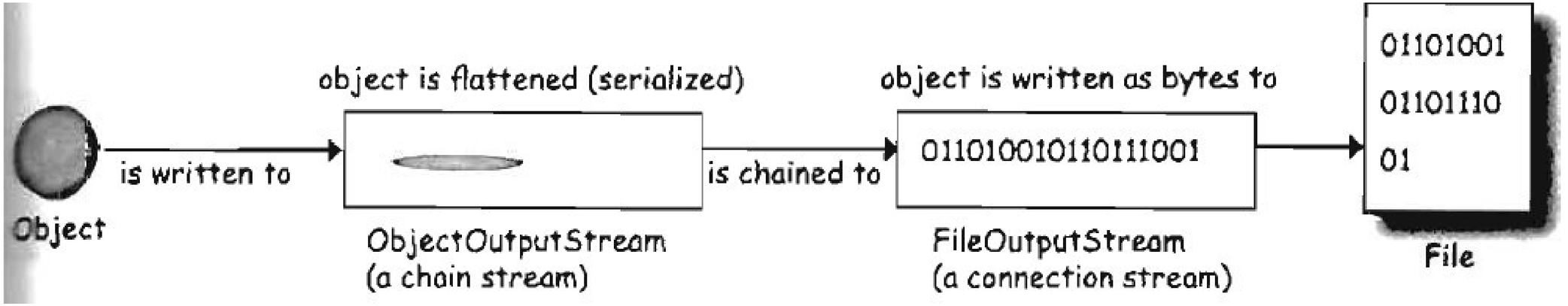

```
try {
    //writing in file
    DataOutputStream dos = new DataOutputStream
            (new FileOutputStream(
            new File("foo.txt")));
    for(int i=0; i<10;i++){
        dos.writeInt(i);
        dos.writeDouble(6.6);
        dos.writeShort(i);
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (Exception ex) {
    ex.printStackTrace();
}
```


rgupta.mtech@gmail.com

Serialization

- **Storing the state of the object on a file along some metadata....so that it Can be recovered back.....**
- **Serialization used in RMI (Remote method invocation) while sending an object from one place to another in network...**



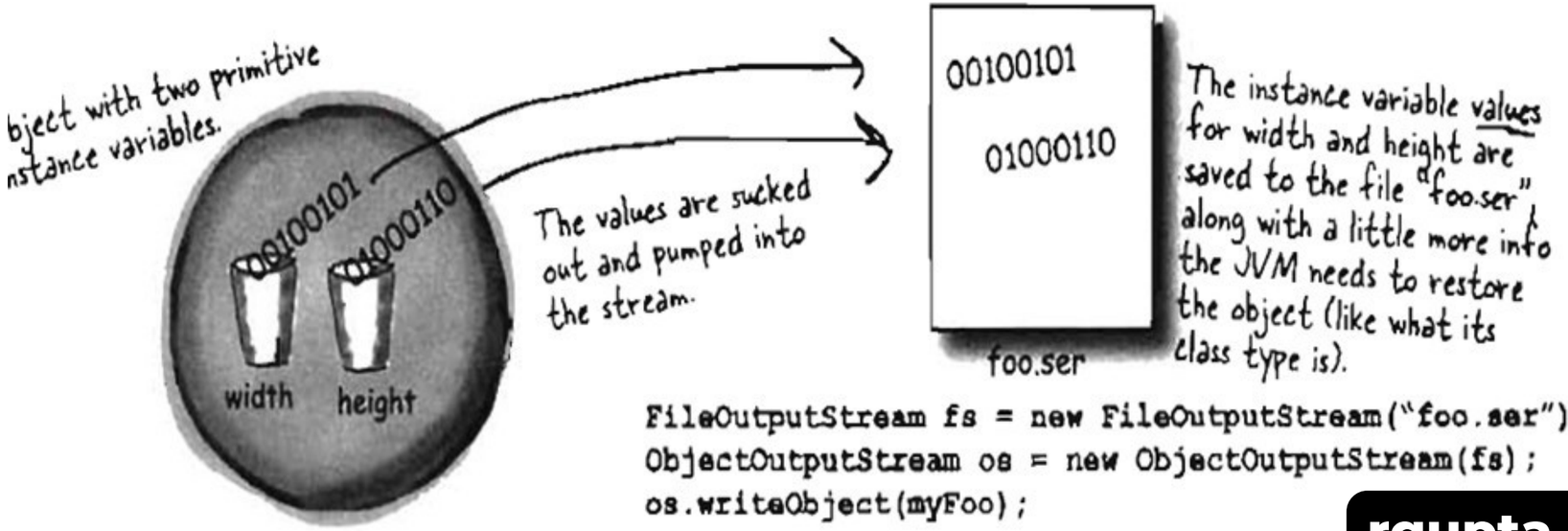


rgupta.mtech@gmail.com

What actually happens during Serialization

The state of the object from heap is sucked and written along with some meta data in an file....

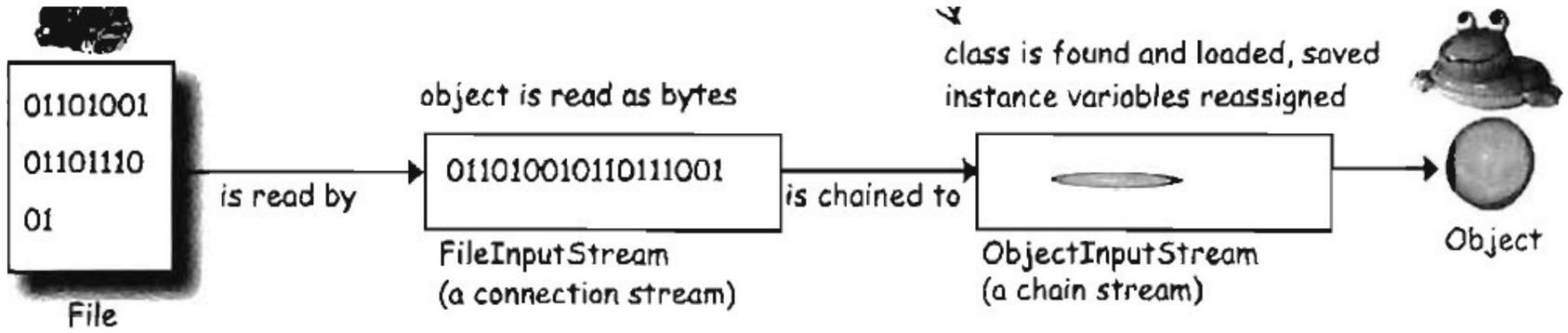
When an object is serialized, all the objects it refers to from instance variables are *also* serialized. And all the objects *those* objects refer to are serialized. And all the objects *those* objects refer to are serialized... and the best




```
FileOutputStream fs = new FileOutputStream("foo.ser");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(myFoo);
```

De- Serialization

- **When an object is de-serialized, the JVM attempts to bring object back to the life by making an new object on the heap that have the same state as original object**
- **Transient variable don’t get saved during serialization hence come with null !!!**



rgupta.mtech@gmail.com

Hello world

```
class Box implements Serializable{
	private static final long serialVersionUID = 1L;
	int l,b;

	public Box(int l, int b) {
		super();
		this.l = l;
		this.b = b;
	}

	public String toString() {
		return "Box [l=" + l + ", b=" + b + "]";
	}

}
```

```
Box box=new Box(22, 33);
```

```
FileOutputStream fo=new FileOutputStream("c:\\raj\\foofoo.ser");
ObjectOutputStream os=new ObjectOutputStream(fo);

os.writeObject(box);
```

```
box=null;//nullify to prove that object come by de-ser...
```

```
FileInputStream fi=new FileInputStream("c:\\raj\\foofoo.ser");
ObjectInputStream oi=new ObjectInputStream(fi);

box=(Box)oi.readObject();
```

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

rgupta.mtech@gmail.com

Serialization vs Externalization

Serialization	Externalization
It is Meant for Default Serialization.	It is Meant for Customized Serialization.
Here Everything Takes Care by JVM and Programmer doesn't have any Control.	Here Everything Takes Care by Programmer and JVM doesn't have any Control.
In Serialization Total Object will be Saved to the File Always whether it is required OR Not.	In Externalization Based on Our Requirement we can Save Either Total Object OR Part of the Object.
Relatively Performance is Low.	Relatively Performance is High.
Serialization is the best choice if we want to save total object to the file.	Externalization is the best choice if we want to save part of the Object to the file.
Serializable Interface doesn't contain any Method. It is a *Marker* Interface.	Externalizable Interface contains 2 Methods, *wrteExternal*() and *readExternal()*. So it is Not Marker Interface.
Serializable implemented Class Not required to contain public No - Argument Constructor.	Externalizable implemented Class should Compulsory contain public No - Argument Constructor. Otherwise we will get Runtime Exception Saying *InvalidClassException*.
transient Key Word will Play Role in Serialization.	transient Key Word won't Play any Role in Externalization. Of Course it is Not Required.

rgupta.mtech@gmail.com

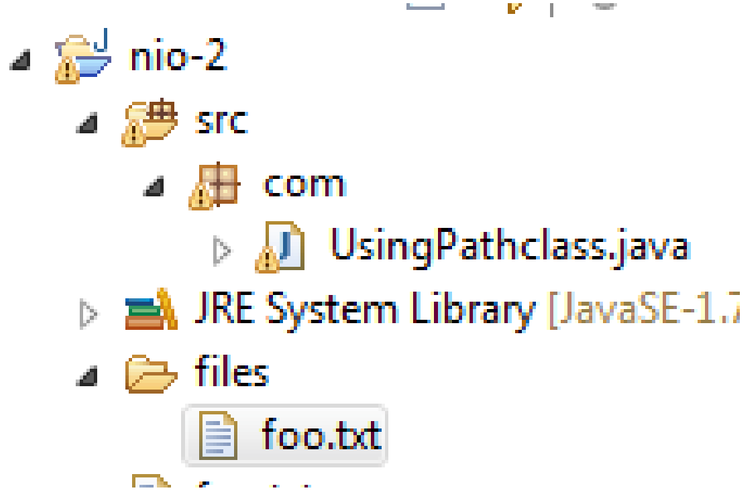
Java NIO-2

1. Using the Path class
2. Managing files and directories
3.Reading and writing text files
4.Reading and writing binary files
4.Walking the directory tree
5. Finding files.
rgupta.mtech@gmail.com
Java Training
6.Watching
a directory
for file changes

rgupta.mtech@gmail.com

Using the Path class

```java
public class UsingPathclass {

    public static void main(String[] args) {
        Path path=Paths.get("files/foo.txt");
        System.out.println(path.toString());
        System.out.println(path.getNameCount());
        System.out.println(path.getFileName());
        System.out.println(path.getName(0));
        System.out.println(path.getName(path.getNameCount()-1));
    }
}
```




```
<terminated> UsingPathclass [Java Ap
files\foo.txt
2
foo.txt
files
foo.txt
```

rgupta.mtech@gmail.com

Managing files and directories CRUD

```
//copy files delete file if already exist.....

 Path source=Paths.get("files/foo.txt");
 Path target=Paths.get("files/temp.txt");
 Files.copy(source, target,StandardCopyOption.REPLACE_EXISTING);

 //Delete.......
 Path todelete=Paths.get("files/temp.txt");
 Files.delete(todelete);
 System.out.println("file deleted.......");

 //creating directory

 Path dir=Paths.get("files/newDir");
 Files.createDirectories(dir);

 //moving an file to dir
 //Files.move(source, target, options)
 Files.move(source, dir.resolve(source.getFileName()), StandardCopyOption.REPLACE_EXISTING);
```

rgupta.mtech@gmail.com

Read/Write text files

```
public static void main(String[] args) {
    Path source=Paths.get("files/old.txt");
    Path target=Paths.get("files/new.txt");

    Charset charset=Charset.forName("US-ASCII");
    try
    (
            BufferedReader br=Files.newBufferedReader(source,charset);
            BufferedWriter wr=Files.newBufferedWriter(target, charset);
    )
    {
        String line=null;
        while((line=br.readLine())!=null)//reading.......
        {
            System.out.println(line);
            wr.append(line,0,line.length());//writing
            wr.newLine();//will append new line

        }
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
```

rgupta.mtech@gmail.com

Reading and writing binary files

```
try
(
        FileInputStream in=new FileInputStream("files/sun.jpg");
        FileOutputStream fs=new FileOutputStream("files/new.jpg");
        //will also create an new file
)
{
    int c;
    while((c=in.read())!=-1)
    {
        fs.write(c);
    }
}
catch(IOException e)
{
    e.printStackTrace();
}
```

rgupta.mtech@gmail.com

Walking the directory tree

Step 1: Create own custom class extending SimpleFileVisitor class SimpleFileVisiter class consist of 4 callback methods

```
class MyFileVisiter extends SimpleFileVisitor<Path>
{

    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
            throws IOException {
        return super.preVisitDirectory(dir, attrs);
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
            throws IOException {
        return super.visitFile(file, attrs);
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc)
            throws IOException {
        return super.visitFileFailed(file, exc);
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc)
            throws IOException {
        return super.postVisitDirectory(dir, exc);
    }
```

rgupta.mtech@gmail.com

Walking the directory tree

```
class MyFileVisiter extends SimpleFileVisitor<Path>
{

    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
            throws IOException {

        System.out.println("About to visit: "+dir);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
            throws IOException {
        if(attrs.isRegularFile())
        {
            System.out.println("Regular file:"+file);
        }
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc)
            throws IOException {
        System.out.println(exc.getMessage());
        return FileVisitResult.CONTINUE;
    }
```

rgupta.mtech@gmail.com

Walking the directory tree

```
    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc)
            throws IOException {
        System.out.println("visited: "+dir);
        return FileVisitResult.CONTINUE;
    }

}
public class WalkingTheDirectoryTree {

    public static void main(String[] args) throws IOException {
        Path fileDir=Paths.get("files2");
        MyFileVisiter visitor=new  MyFileVisiter();
        Files.walkFileTree(fileDir, visitor);
    }
}
```

rgupta.mtech@gmail.com

Finding an particular file/dir

Step 1: Create FileFinder class that extends SimpleFileVisitor<Path>

Step 2: Create PathMatcher

```
class FileFinder extends SimpleFileVisitor<Path>{
    PathMatcher matcher;
    public ArrayList<Path>foundPath=new ArrayList<>();

    FileFinder(String pattern){
        matcher=FileSystems.getDefault().getPathMatcher("glob:"+pattern);
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
            throws IOException {
        Path name=file.getFileName();
        System.out.println("Examining :"+name);
        if(matcher.matches(name))
        {
            foundPath.add(file);
        }
        return FileVisitResult.CONTINUE;
    }
```

rgupta.mtech@gmail.com

Finding an particular file/dir

```
Path fileDir=Paths.get("files2");
//looks for specific file
FileFinder finder= new FileFinder("file1.txt");

Files.walkFileTree(fileDir, finder);

ArrayList<Path>foundFilles=finder.foundPath;

if(foundFilles.size()>0)
{
    for(Path path:foundFilles)
    {
        System.out.println(path.toRealPath(LinkOption.NOFOLLOW_LINKS));
    }
}
```

rgupta.mtech@gmail.com

Watching a directory for file changes

- **Monitoring Update/delete modify any file/directory in file system....**
- **Watch service**
  - **Service that monitor an directory and report when any content in that directory is deleted/modified etc**

rgupta.mtech@gmail.com

Watching a directory for file changes

```
try(
        WatchService service=FileSystems.getDefault().newWatchService();
    )
    {
    Map<WatchKey,Path>keyMap=new HashMap<>();
    Path path=Paths.get("files2");

    keyMap.put(path.register(service, StandardWatchEventKinds.ENTRY_CREATE,
            StandardWatchEventKinds.ENTRY_DELETE,StandardWatchEventKinds.ENTRY_MODIFY), path);

    WatchKey watchKey;

        do{

            watchKey=service.take();

            Path eventDir=keyMap.get(watchKey);
            for(WatchEvent<?>event:watchKey.pollEvents())
            {
                WatchEvent.Kind<?>kind=event.kind();

               Path eventPath=(Path)event.context();
               System.out.println(eventDir +" :"+kind+" :"+eventPath);
            }

      }while(watchKey.reset());

    }

catch(Exception ex){

}
```

rgupta.mtech@gmail.com

Session 7:
Java Collection API

- Collections Framework introduction
- List, Set, Map
- Iterator, ListIterator and Enumeration
- Collections and Array classes
- Sorting and searching, Comparator vs Comparable
- Generics, wildcards, using extends and super, bounded type
- Hands on & Lab

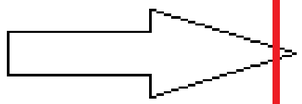
rgupta.mtech@gmail.com

Object

- **Object is an special class in java defined in java.lang**
- **Every class automatically inherit this class whether we say it or not...**

We Writer like...

```
class Employee{
        int id;
        double salary;
        ....
        ....
        }
```



Java compiler convert it as...

```
class Employee extends Object{
        int id;
        double salary;
        ....
        ....
}
```

Why Java has provided this class?

rgupta.mtech@gmail.com

Method defined in Object class

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

toString( )

If we do not override toString() method of Object class it print Objec Identification number by default

We can override it to print some useful information....

```
@Override
public String toString() {
    return "Employee [id=" + id + ", salary=" + salary + "]";
}
```

```
class Employee{
    private int id;
    private double salary;

    public Employee(int id, double salary) {
        this.id = id;
        this.salary = salary;
    }

}
```

```
.....
.....
Employee e=new Employee(22, 333333.5);
System.out.println(e);
.....
....
```

O/P

---

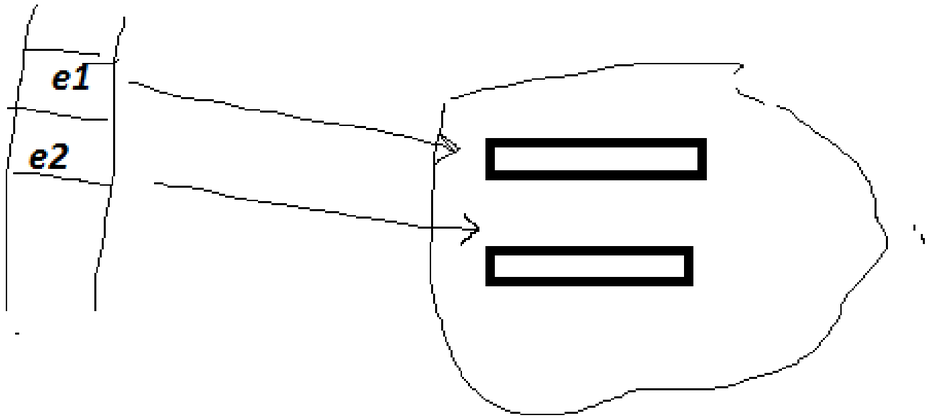
com.Employee@addbf1

Java simply print object identification number not so useful message for client

rgupta.mtech@gmail.com

equals

```
Employee e1=new Employee(22, 333333.5);
         Employee e2=new Employee(22, 333333.5);

         if(e1==e2)
              System.out.println("two employees are equals....");
         else
              System.out.println("two employees are not equals....'
```



O/P would be two employees are not equals.... ???
Problem is that using == java compare object id of two object and that can never be equals,so we are getting meaningless result...

rgupta.mtech@gmail.com

Overriding equals()

```
@Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        if (id != other.id)
            return false;
        if (Double.doubleToLongBits(salary) != Double
                .doubleToLongBits(other.salary))
            return false;
        return true;
    }
```

rgupta.mtech@gmail.com

hashCode()

- **Whenever you override equals()for an type don’t forget to override hashCode() method…**
  - **hashCode() make DS efficient What hashCode does HashCode divide data into buckets**
  - **Equals search data from that bucket…**

```
@Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        long temp;
        temp = Double.doubleToLongBits(salary);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        return result;
    }
```

rgupta.mtech@gmail.com

clone()

- **Lets consider an object that creation is very complicated, what we can do we can make an clone of that object and use that Costly , avoid using cloning if possible, internally depends on serialization**
- **Must make class supporting cloning by implementing an marker interface ie Cloneable**

```
class Employee implements Cloneable{
    private int id;
    private double salary;

    public Employee(int id, double salary) {
        this.id = id;
        this.salary = salary;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // TODO Auto-generated method stub
        return super.clone();
        //can write more code
    }


}
```

rgupta.mtech@gmail.com

finalize()

As you are aware ..Java don’t support destructor Programmer is free from memory management
Memory mgt is done by an component of JVM ie called Garbage collector GC GC runs as low priority thread..
We can override finalize() to request java Please run this code before recycling this object”
Cleanup code can be written in finalize() method Not reliable, better not to use...

Demo programm

- **WAP to count total number of employee object in the memory at any moment of time if an object is nullified then reduce count....**

rgupta.mtech@gmail.com

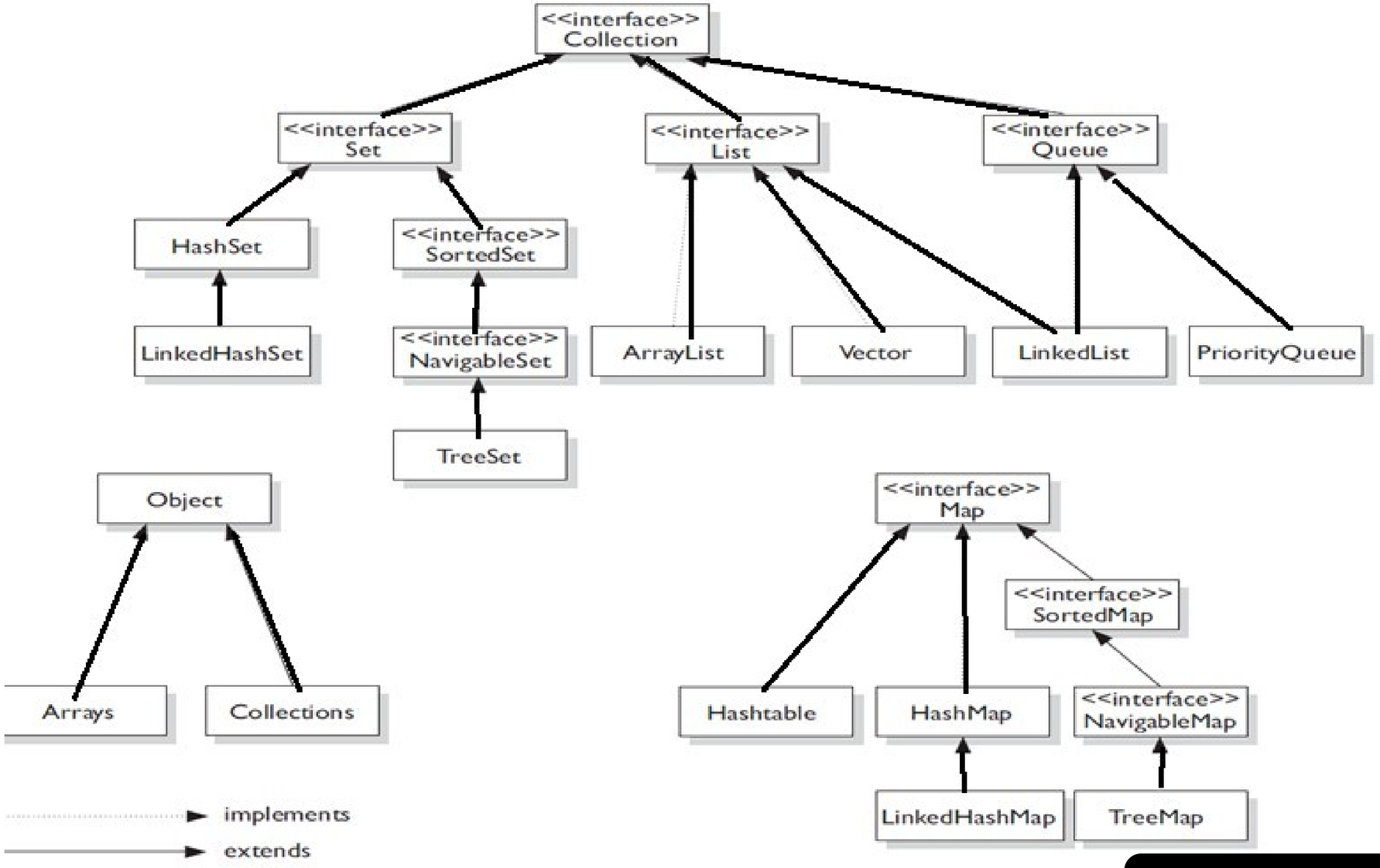
Java collection

Java collections can be considered a kind of readymade data structure, we should only need to know how to use them and how they work....

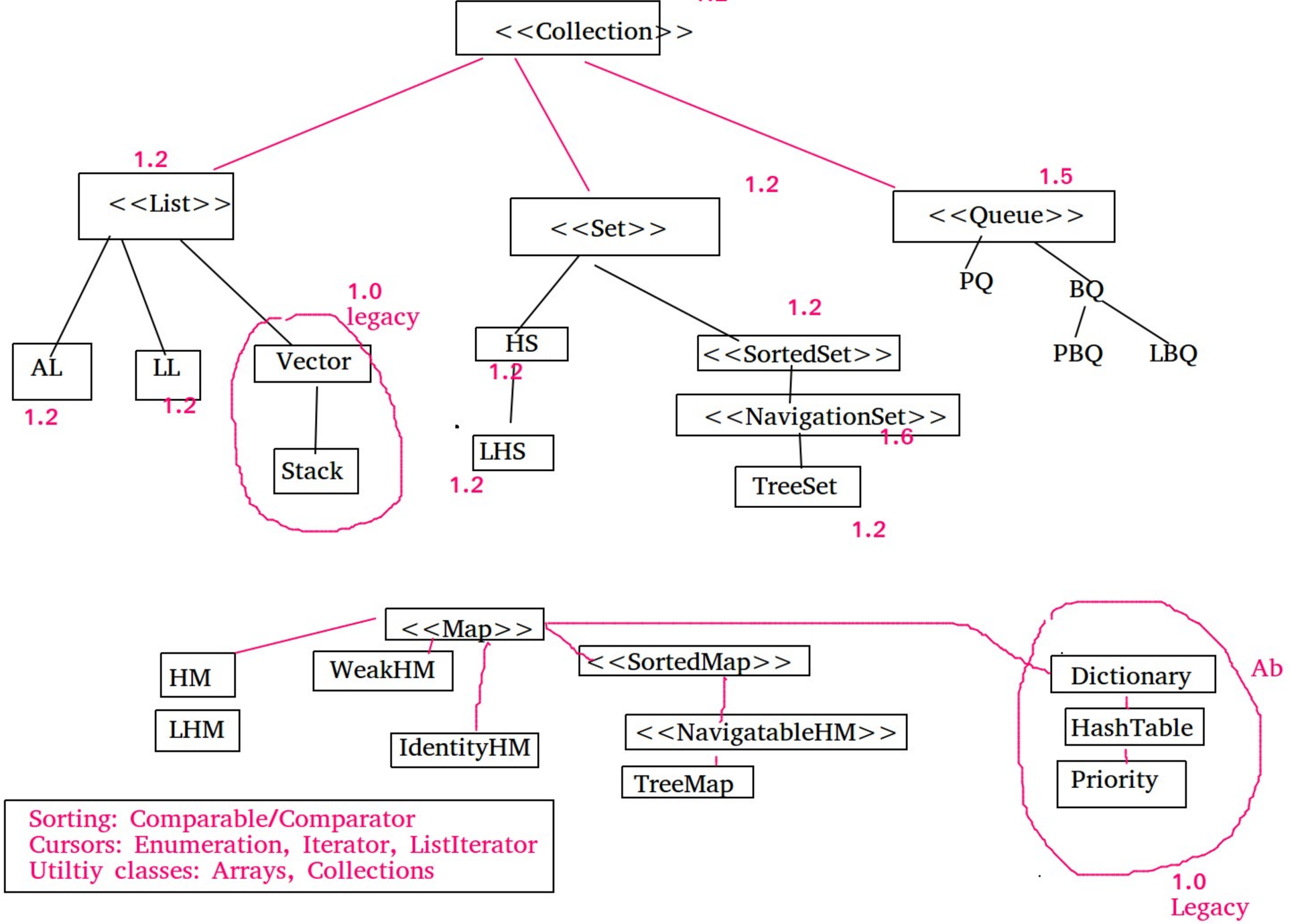
collection: Name of topic
Collection : Base interface
Collections: Static utility class provide various useful algorithm

rgupta.mtech@gmail.com

collection



rgupta.mtech@gmail.com


<<Collection>>
1.2
<<List>>
1.2
<<Set>>
1.5
<<Queue>>
AL
1.2
LL
1.2
Vector
1.0
legacy
Stack
HS
1.2
LHS
1.2
<<SortedSet>>
<<NavigationSet>>
1.6
TreeSet
1.2
PQ
BQ
PBQ
LBQ
<<Map>>
HM
LHM
WeakHM
IdentityHM
<<SortedMap>>
<<NavigatableHM>>
TreeMap
Dictionary
HashTable
Priority
1.0
Legacy
Ab
Sorting: Comparable/Comparator
Cursors: Enumeration, Iterator, ListIterator
Utiltiy classes: Arrays, Collections

Four type of collections

Collections come in four basic flavors:

- **Lists** *Lists* of things (classes that implement List).
- **Sets** *Unique* things (classes that implement Set).
- **Maps** Things with a *unique* ID (classes that implement Map).
- **Queues** Things arranged by the order in which they are to be processed.

rgupta.mtech@gmail.com

Iterator in Java

```
List<String>list=new LinkedList<String>();
list.add("a");
list.add("b");

ListIterator<String> it = list.listIterator();
while(it.hasNext()){
    String val=it.next();
    if(val.equals("raj"))
        it.remove();
    else if(val.equals("a"))
        it.add("aa");
    else if(val.equals("b"))
        it.set("b1");
}
```

rgupta.mtech@gmail.com

ArrayList:aka growable array

```
List<String>list=new ArrayList<String>();
...
...
list.size();
list.contains("raj");

test.remove("hi");

Collections.sort(list);
```

Note:

```
Collections.sort(list,Collections.reverseOrder());

Collections.addAll(list2,list1);
```

Add all elements from list1 to end of list2

```
Collections.frequency(list2,"foo");
```

---

print frequency of "foo" in the list2 collection

```
boolean flag=Collections.disjoint(list1,list);
```

---

return "true" if nothing is common in list1 and list2

Sorting with the Arrays Class

```
Arrays.sort(arrayToSort)

Arrays.sort(arrayToSort, Comparator)
```

rgupta.mtech@gmail.com

ArrayList of user defined object

```
class Employee{
        int id;
        float salary;
        //getter setter
        //const
        //toString
        }

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

        list.add(new Employee(121,"rama");
        list.add(new Employee(121,"rama");
        list.add(new Employee(121,"rama");

        System.out.println(list);

        Collections.sort(list);
```

How java can decide how to sort?

rgupta.mtech@gmail.com

Comparable and Comparator interface

We need to teach Java how to sort user define object
Comparable and Comparator interface help us to tell java how to sort user define object....

Comparable	Comparator
java.lang	java.util
Natural sort	seconday sorts
Only one sort sequence is possible	as many as you want
need to change the design of the class	Dont need to change desing of the class
need to override	need to override
public int compareTo(Employee o)	public int compare(Employee o1, Employee o2)

rgupta.mtech@gmail.com

Implementing Comparable

```
class Employee implements Comparable<Employee>{
    private int id;
    private double salary;
    .......
    .......
    .......

        @Override
    public int compareTo(Employee o) {
        // TODO Auto-generated method stub
        Integer id1=this.getId();
        Integer id2=o.getId();
        return id1.compareTo(id2);
    }
```

rgupta.mtech@gmail.com

Comparator

Don’t need to change Employee class

```
class SalarySorter implements Comparator<Employee>{

    @Override
    public int compare(Employee o1, Employee o2) {
        // TODO Auto-generated method stub
        Double sal1=o1.getSalary();
        Double sal2=o2.getSalary();

        return sal1.compareTo(sal2);
    }

}
```

rgupta.mtech@gmail.com

Useful stuff

Converting Arrays to Lists

```
String[] sa = {"one", "two", "three", "four"};
List sList = Arrays.asList(sa);
```

Converting Lists to Arrays

```
List<Integer> iL = new ArrayList<Integer>();

for(int x=0; x<3; x++)
iL.add(x);

Object[] oa = iL.toArray(); // create an Object array

Integer[] ia2 = new Integer[3];

ia2 = iL.toArray(ia2); // create an Integer array
```

Arrays.binarySearch(arrayFromWhichToSearch,"to search"))

---

return -ve no if no found

array must be sorted before hand otherwise o/p is not predictiale

user define funtion to remove stuff from a arraylist /linkedlist

---

```
removeStuff(list,2,5);


...
...

public void removeStuff(List<String>l, int from, int to)
{
        l.subList(from,to).clear();
}
```

rgupta.mtech@gmail.com

Useful examples

Merging two link lists

```
ListIterator ita=a.listItrator();
Iteratory itb=b.iterator();

while(itb.hasNext())
{
        if(ita.hasNext())
                ita.next();

        ita.add(itb.next());
}
```

Removing every second element from an linkedList

```
itb=b.iterator();

while(itb.hasNext())
{
        itb.next();

        if(itb.hasNext())
        {
                itb.next();

                itb.remove();
        }
}
```

rgupta.mtech@gmail.com

LinkedList

AKA Doubly Link list...can move back and forth

Imp methods
---

boolean hasNext()
Object next()
boolean hasPrevious()
Object previous()

More methods
---

void addFirst(Object o);

void addLast(Object o);

object getFirst();

object getLast();

add(int pos,Object o);

rgupta.mtech@gmail.com

ArrayList vs LinkedList

Java implements ArrayList as array internally

- **Hence good to provide starting size**
- **i.e. List<String> s=new ArrayList<String>(20); is better then List<String> s=new ArrayList<String>();**
- **Removing element from starting of arraylist is very slow?**
- **list.remove(0);**
- **if u remove first element, java internally copy all the element (shift by one)**
- **Adding element at middle in ArrayList is very inefficient...**

rgupta.mtech@gmail.com

Performance Array List vs LinkedList

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class App
{
 public static void main(String[] args)
          {
          List<Integer> arrayList = new ArrayList<Integer>();
          List<Integer> linkedList = new LinkedList<Integer>();

           doTimings("ArrayList", arrayList);
           doTimings("LinkedList" , linkedList);
    }
```

```
private static void doTimings(String type, List<Integer> list)
  {
     for(int i=0; i<1E5; i++)
            list.add(i);
     long start = System.currentTimeMillis();

     /*
     // Add items at end of list
     for(int i=0; i<1E5; i++)
          {
       list.add(i);
     }
     */

     // Add items elsewhere in list
     for(int i=0; i<1E5; i++)
          {
       list.add(0, i);
     }
     long end = System.currentTimeMillis();

     System.out.println("Time taken: " + (end - start) + " ms for " + type);
   }
}
```

```
Time taken: 7546 ms for ArrayList
Time taken: 76 ms for LinkedList
```

rgupta.mtech@gmail.com

HashMap

Key ---->Value declaring an hashmap

- **HashMap<Integer, String> map = new HashMap<Integer, String>();**

Populating values

```
map.put(5, "Five");
map.put(8, "Eight");
map.put(6, "Six");
map.put(4, "Four");
map.put(2, "Two");

String text = map.get(6);

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

rgupta.mtech@gmail.com

HashMap Internal Structure

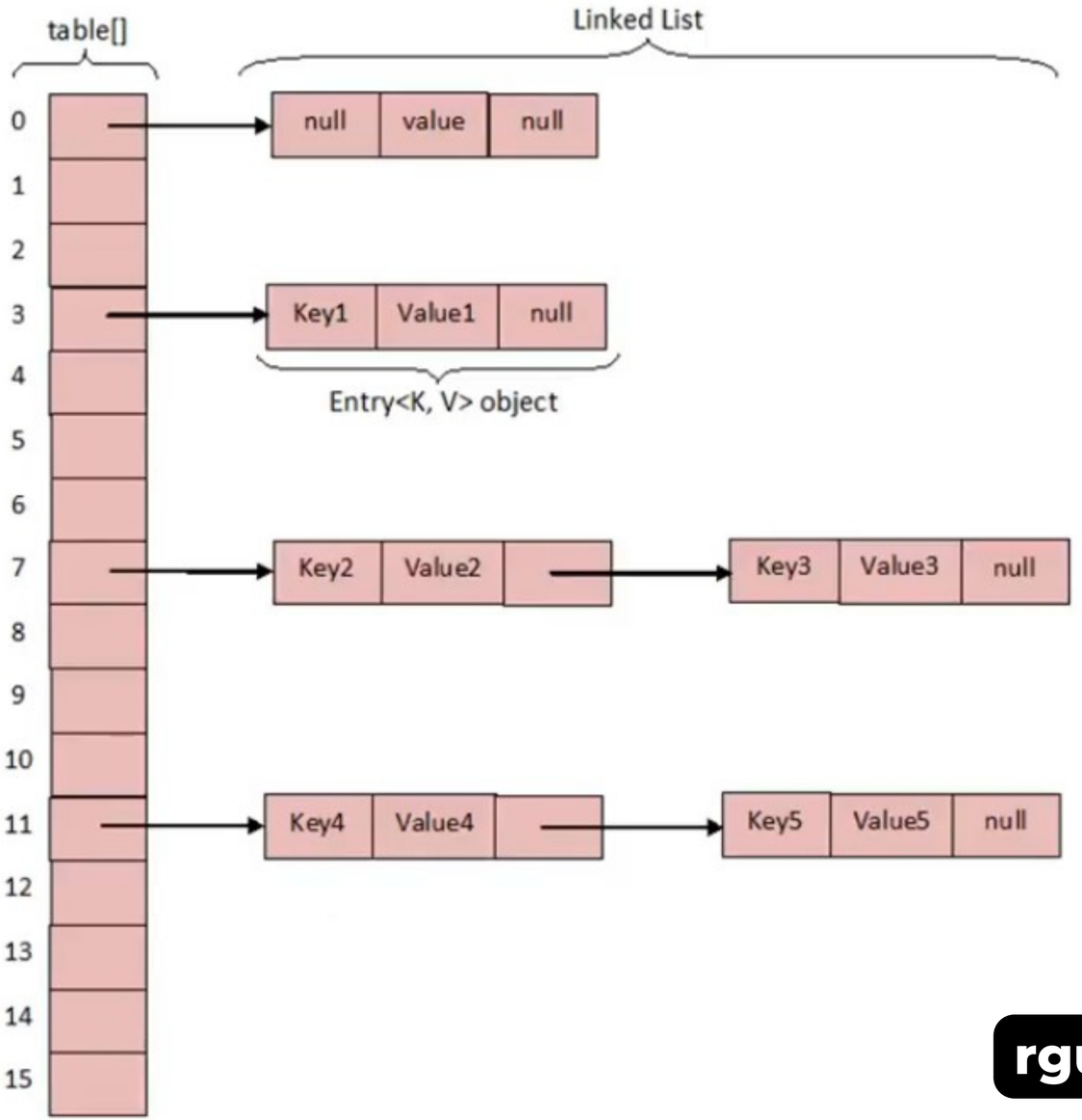
- HashMap stores the data in the form of key-value pairs.
- Each key-value pair is stored in an object of Entry<K, V> class.

```
static class Entry<K,V> implements Map.Entry<K,V>
{
        final K key;
        V value;
        Entry<K,V> next;
        int hash;

}
```

- **key** : It stores the key of an element and its final.
- **value** : It holds the value of an element.
- **next** : It holds the pointer to next key-value pair. This attribute makes the key-value pairs stored as a linked list.
- **hash** : It holds the hashcode of the key.

rgupta.mtech@gmail.com

HashMap Internal Structure



rgupta.mtech@gmail.com

HashMap Internal Structure

Map score=new HashMap<String,Integer>();

Score.put("Kohli",100)

Hash(kohli) -> 45612

indexFor(hash code) ->3

Score.put("Sachin",200)

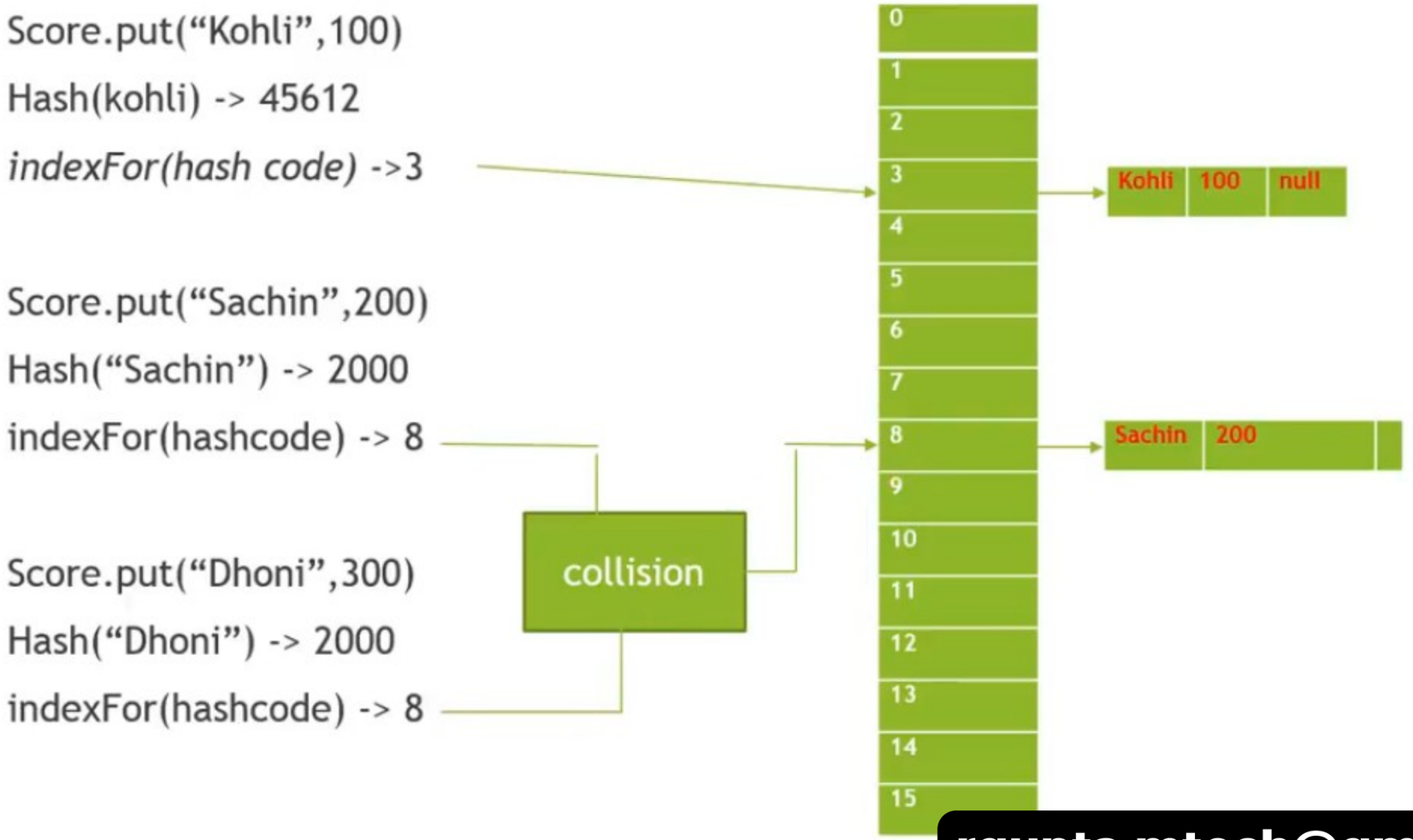
Hash("Sachin") -> 2000

indexFor(hashcode) -> 8

Score.put("Dhoni",300)

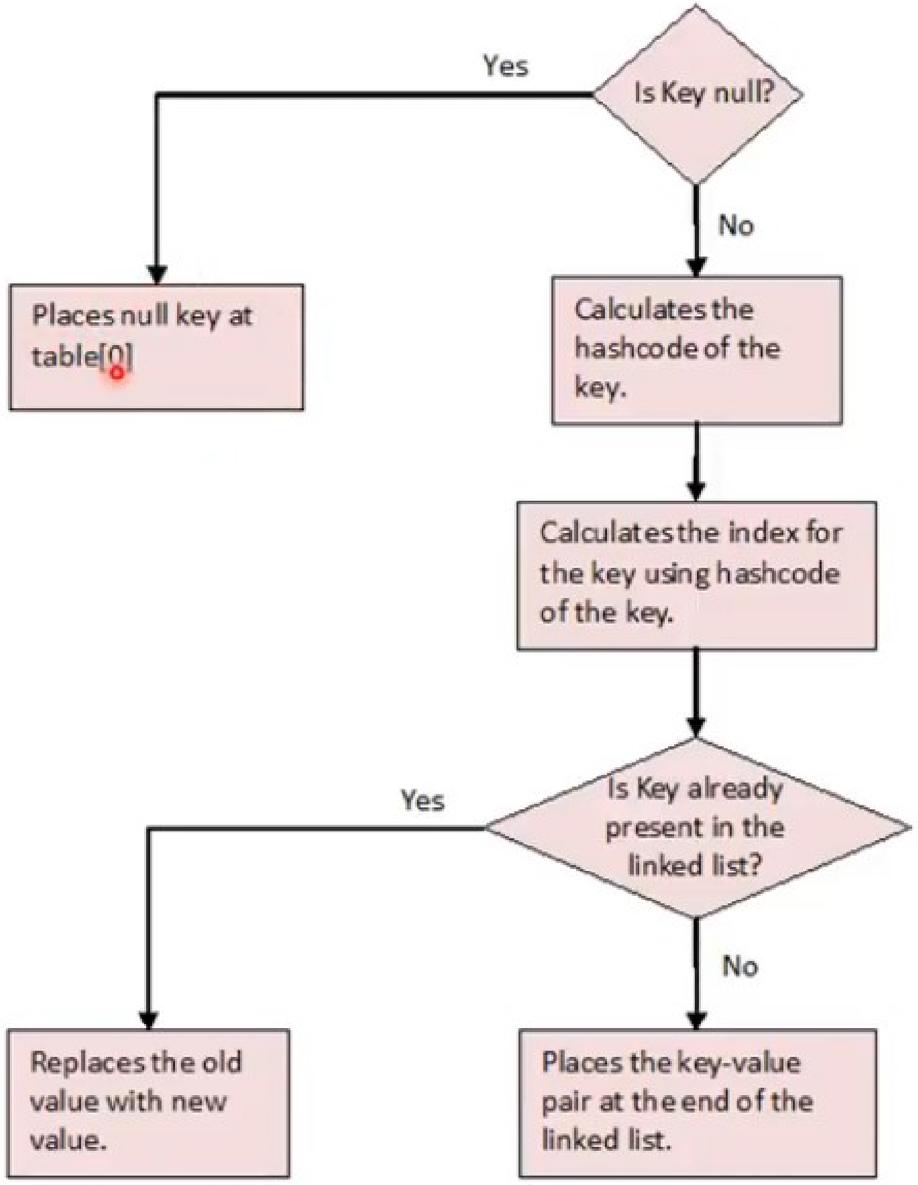
Hash("Dhoni") -> 2000

indexFor(hashcode) -> 8



rgupta.mtech@gmail.com

How put() method works?



- In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.
- If key given already exist in HashMap, the value is replaced with new value.
- hash code of null key is 0.
- When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

rgupta.mtech@gmail.com

How get() method works?

How get() method Works?

Score.get("Dhoni")

Calculate hash code of Key {"Dhoni"}. It will be generated as 2000.

Calculate index by using index method it will be 8.

Go to index 8 of array and compare first element's key with given key. If both are equals then return the value, otherwise check for next element if it exists.

In our case it is found as second element and returned value is 300.

rgupta.mtech@gmail.com

User define key in HashMap

If you are using user define key in HashMap do not forget to override hashcode for that class
Why?
We may not find that content again !

HashMap vs Hashtable

- **Hashtable is threadsafe, slow as compared to HashMap**
- **Better to use HashMap**
- **Some more intresting difference**
- **Hashtable give runtime exception if key is “null” while HashMap**
- **don’t**

rgupta.mtech@gmail.com

Set

three types:

---

hashset
linkedhashset
treeset

HashSet does not retain order.

```
Set<String> set1 = new HashSet<String>();
```

LinkedHashSet remembers the order you added items in

```
Set<String> set1 = new LinkedHashSet<String>();
```

TreeSet sorts in natural order

```
Set<String> set1 = new TreeSet<String>();
```

rgupta.mtech@gmail.com

PriorityQueue

```
public class DemoPQ {
    public static void main(String[] args) {
        PriorityQueue<String> queue=new PriorityQueue<String>();
        queue.add("Amit");
        queue.add("Vijay");
        queue.add("Karan");
        queue.add("Jai");
        queue.add("Rahul");  //same as offer
        //retrieved not remove, throw exp
        System.out.println("head:"+queue.element());
        //retrieved not remove , return null
        System.out.println("head:"+queue.peek());

        System.out.println("iterating the queue elements:");
        Iterator<String> itr=queue.iterator();
        while(itr.hasNext()){
        System.out.println(itr.next());
        }
        //remove from head, throws ex if empty
        System.out.println(queue.remove());
        //remove from head, return null if empty
        System.out.println(queue.poll());

    }
}
```

rgupta.mtech@gmail.com

Need of Generics

Before Java 1.5

list=new ArrayList();
Can add anything in that list, Problem while retrieving

Now Java 1.5 onward
List<String>
list=new ArrayList<String>();
list.add(“foo”);//ok
list.add(22);// compile time error
Generics provide type safety
Generics is compile time phenomena…

rgupta.mtech@gmail.com

Caution with Generics

Try not to mix non Generics code and Generics code...we can have strange behaviour.

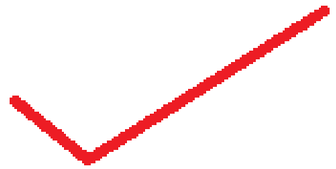

```
package com;
import java.util.*;

public class DemoGen1 {
    public static void main(String[] args) {

        List<String> list=new ArrayList<String>();
        list.add("foo");
        list.add("bar");

        strangMethod(list);

        for(String temp:list)
            System.out.println(temp);
    }

    private static void strangMethod(List list) {
        list.add(new Integer(22));// OMG.....
    }

}
```

rgupta.mtech@gmail.com

Polymorphic behaviour

```
class Animal {
}

class Cat extends Animal{
}

class Dog extends Animal{
     }
```

```
Animal []aa=new Cat[4];// allowed

List<Animal>list=new ArrayList<Cat>()
```



rgupta.mtech@gmail.com

<? extends XXXXXX>

```
package com;
import java.util.*;

public class DemoGen1 {
    public static void main(String[] args) {

        List<Integer> list=new ArrayList<Integer>();
        list.add(22);
        list.add(33);

        strangMethod(list);

        for(Integer temp:list)
            System.out.println(temp);
    }

    private static void strangMethod(List<? extends Number> list) {
        list.add(new Integer(22));//Compile time error..... Good
    }

}
```

in strangMethod() we can pass any derivative of Number class but we are not allowed to modify the list

rgupta.mtech@gmail.com

<? Super XXXXX>

```
class Animal {
}

class Cat extends Animal{
}

class Dog extends Animal{
     }

class CostlyDog extends Dog{
}
     ........
     .........
          List<Dog>list=new ArrayList<Dog>();
          list.add(new Dog("white"));
          list.add(new Dog("red"));
          list.add(new Dog("black"));
          strangMethod(list);

     private static void strangMethod(List<? super Dog> list) {
          list.add(new CostlyDog());
     }

     .........
     ......
```

Anytype of Dog is allowed and can also modify list

rgupta.mtech@gmail.com

Generic class

```
class MyObject<T>{
     T myObject;

     public T getMyObject() {
          return myObject;
     }

     public void setMyObject(T myObject) {
          this.myObject = myObject;
     }

}
public class GenClass {

          public static void main(String[] args) {
               MyObject<String> o=new MyObject<String>();
               o.setMyObject(new Integer(22));

               //System.out.println(it.intValue());
          }
}
```

will not compile !!!

rgupta.mtech@gmail.com

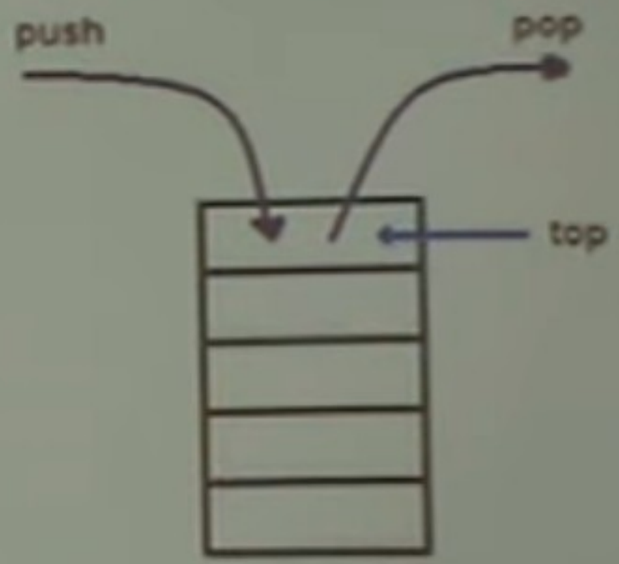
Generaic method

```
class MaxOfThree{

     public static <T extends Comparable<T>> T maxi(T a,T b, T c){
          T max=a;
          if(b.compareTo(a)>0)
               max=b;
          if(c.compareTo(max)>0)
               max=c;
          return max;
     }

}
```

rgupta.mtech@gmail.com

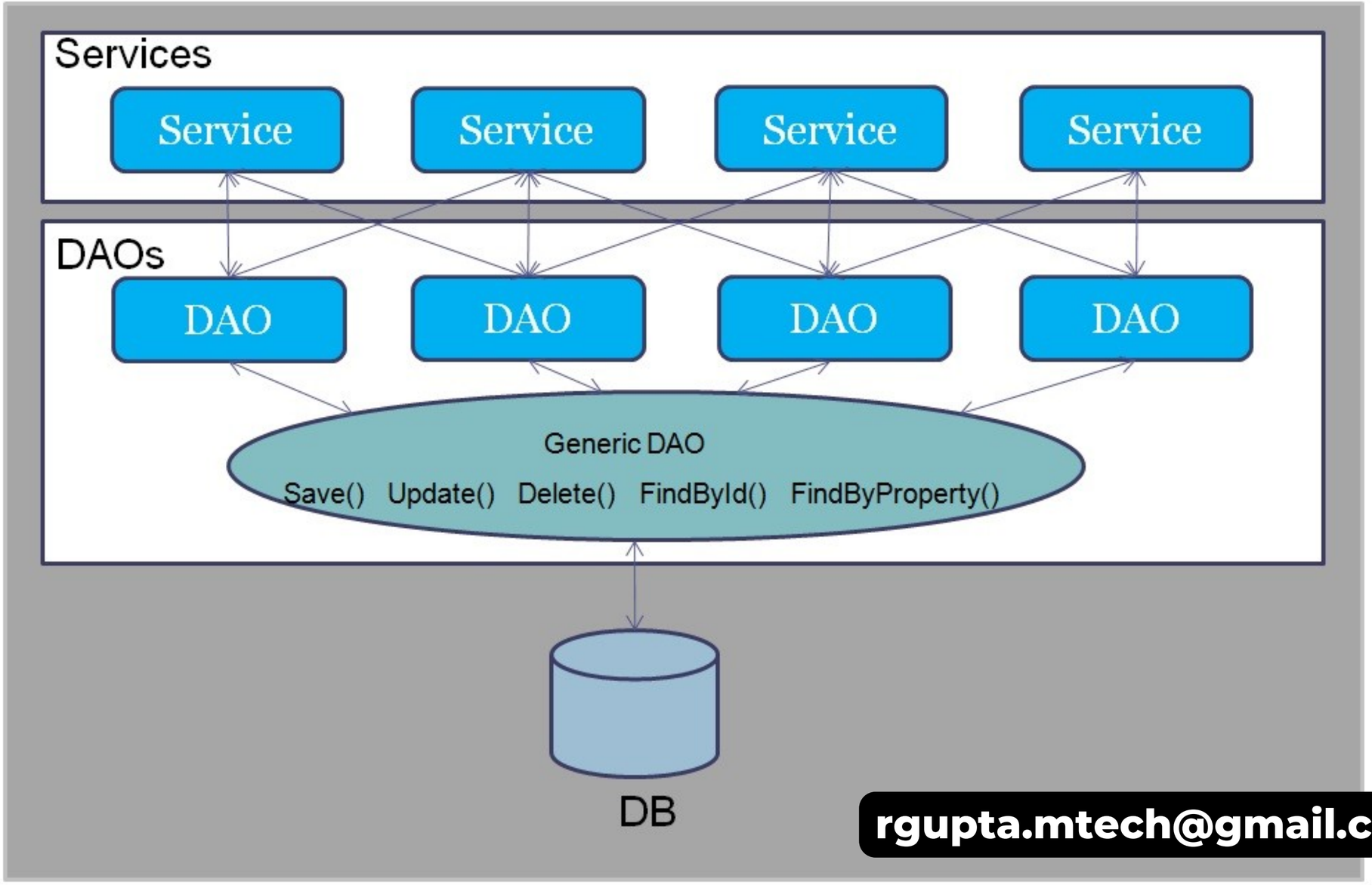
Generaic Stack

```
class Stack<E>
{
  private final int size;
  private int top;
  private E[] elements;
  public Stack() { this(10); }
  public Stack(int s) {
     size = s > 0 ? s : 10;
     top = -1;
     elements = (E[]) new Object[size]; // create array
  }
  public void push(E pushValue) {
     if (top < size - 1) // if stack is not full
     elements[++top] = pushValue; // place pushValue on Stack
  }
  public E pop() {
     if (top > -1) // if stack is not empty
     return elements[top--]; // remove and return top element of Stack
  }
}
```



rgupta.mtech@gmail.com

Generaic Dao Layer



rgupta.mtech@gmail.com