CORE JAVA MASTERY (FOUNDATIONS) 2026

Rajeev Gupta

rgupta.mtech@gmail.com

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

Trainer's Profile

- Senior Corporate Trainer & Technical Consultant with 21+ years helping global engineering teams master Java, Spring Boot, Microservices, AWS Cloud, DevOps, and GenAI for Java.

I specialize in enabling teams to build scalable, observable, production-ready microservices using real-world architecture patterns, hands-on labs, and deep technical coaching.

Technical Expertise

• Java 8–25 — Streams, Concurrency, JVM Internals
• Spring Boot Ecosystem — Security, Data, Spring Cloud
• Microservices Architecture — API Design, SAGA, CQRS, resilience, observability
• Event Streaming — Kafka, RabbitMQ
• AWS Cloud-Native Development — Built & deployed Spring Boot microservices using ECS, EKS, API Gateway, SQS/SNS, with CI/CD via CodePipeline, CodeBuild, GitHub Actions, Terraform
• Containers & DevOps — Docker, Kubernetes (EKS), IaC, CI/CD pipelines
• GenAI for Java Developers — LangChain4J, Spring AI, RAG, Vector DBs, Java-based AI Agents

Corporate Client

- Bank Of America
- MakeMyTrip
- GreatLearning
- Deloitte
- Kronos
- Yamaha Moters
- IBM
- Sapient
- Accenture
- Airtel
- Gemalto
- Cyient Ltd
- Fidelity Investment Ltd
- Blackrock
- Mahindra Comviva
- Iris Software
- harman
- Infosys
- Espire
- Steria
- Incedo
- Capgemini
- HCL
- CenturyLink
- Nucleus
- Ericsson
- Ivy Global
- Avaya
- NEC Technologies
- A.T. Kearney
- UST Global
- TCS
- North Shore Technologies
- Incedo
- Genpact
- Torry Harris
- Indian Air force
- Indian railways

rgupta.mtech@gmail.com

Day 1:
Object Orientation Fundamentals
Basic OOPs, Inheritance, Overriding, Overloading, Polymorphism
Day 2:
Advance Object Orientation, Abstract class, Interface, SOLID Introduction
String, Wrapper classes, Exception Handling, IO
Day 3:
Introduction to Collection API, Multithreading, GOF Introduction
Day 4:
DevOps Introduction, Tools, JDBC, Clean Code
Day 5:
Java 8 Stream API, Optional, Date and Time API

rgupta.mtech@gmail.com

Day 1:
Object Orientation Fundamentals
Basic OOPs, Inheritance, Overriding, Overloading, Polymorphism

rgupta.mtech@gmail.com

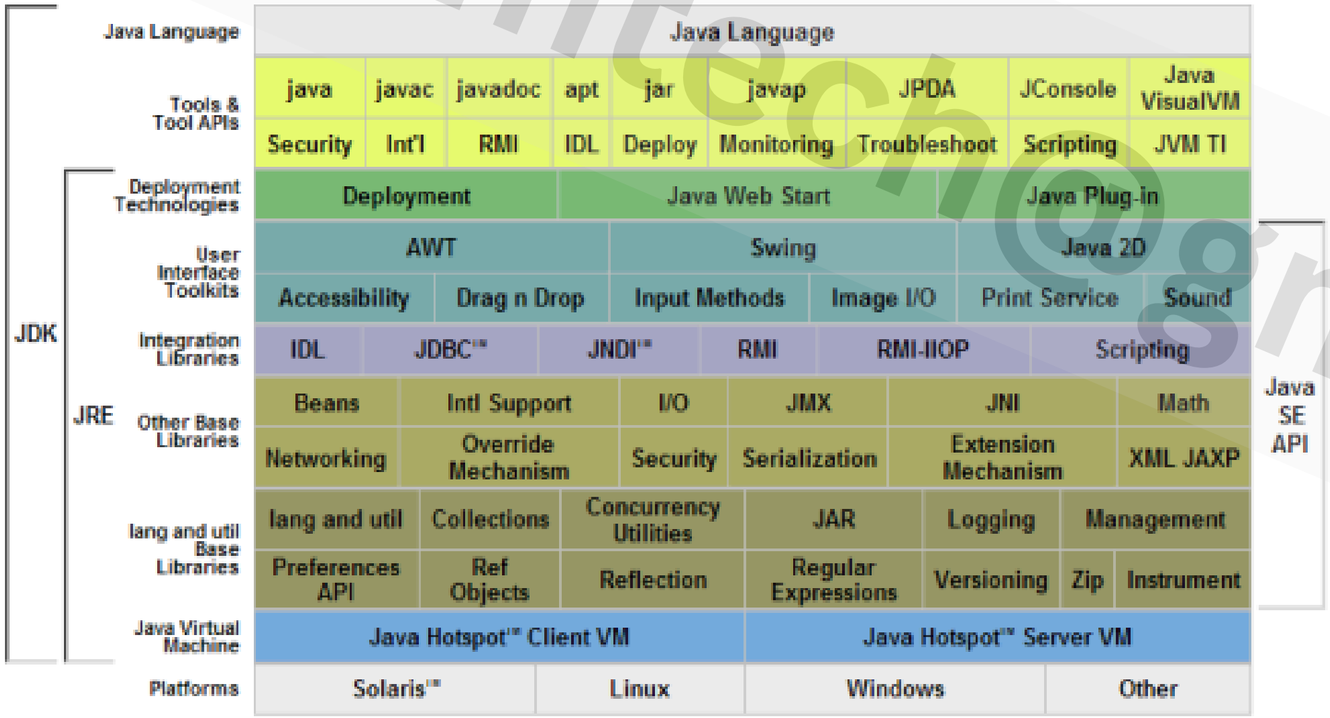
Day-1
Session -1
Introduction

rgupta.mtech@gmail.com

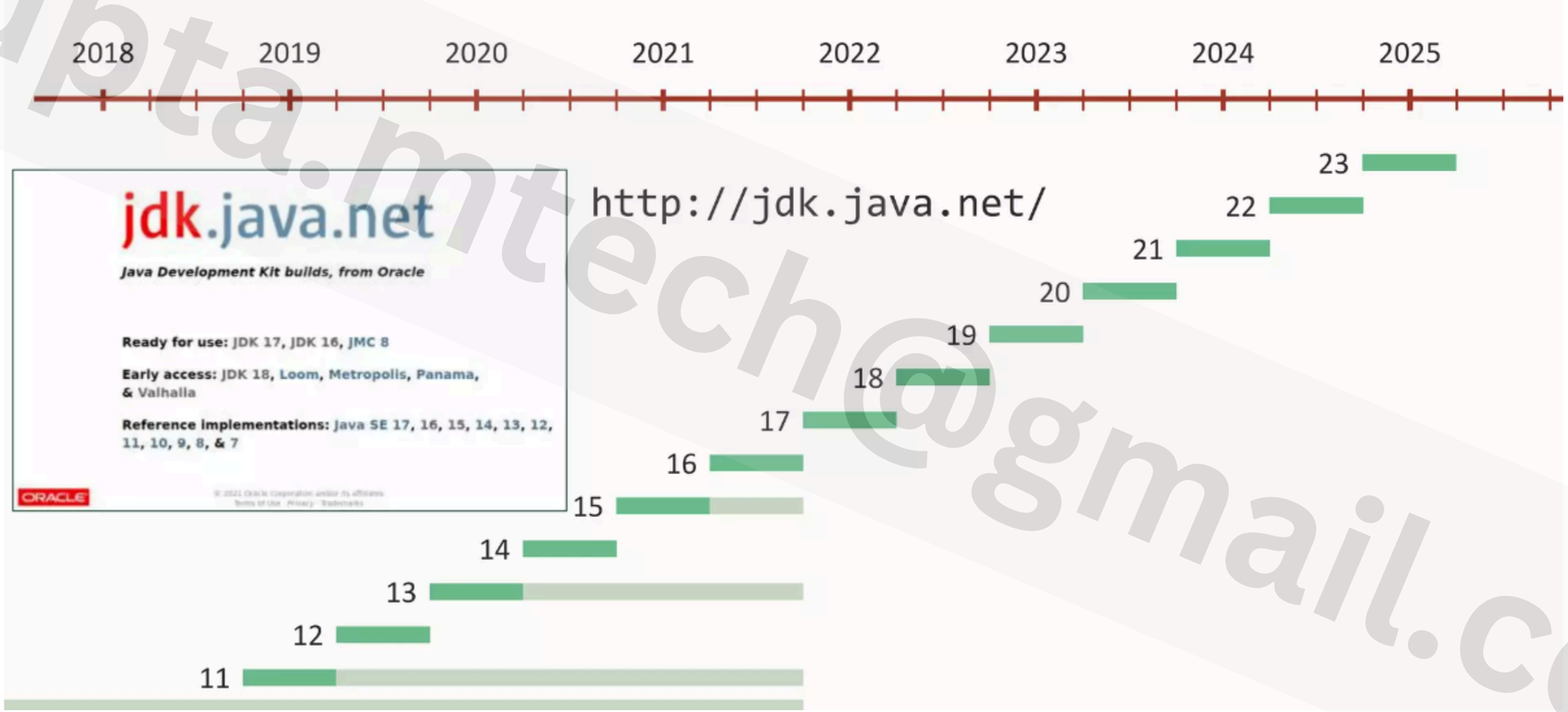
What is Java

Java=OOPL+JVM+lib

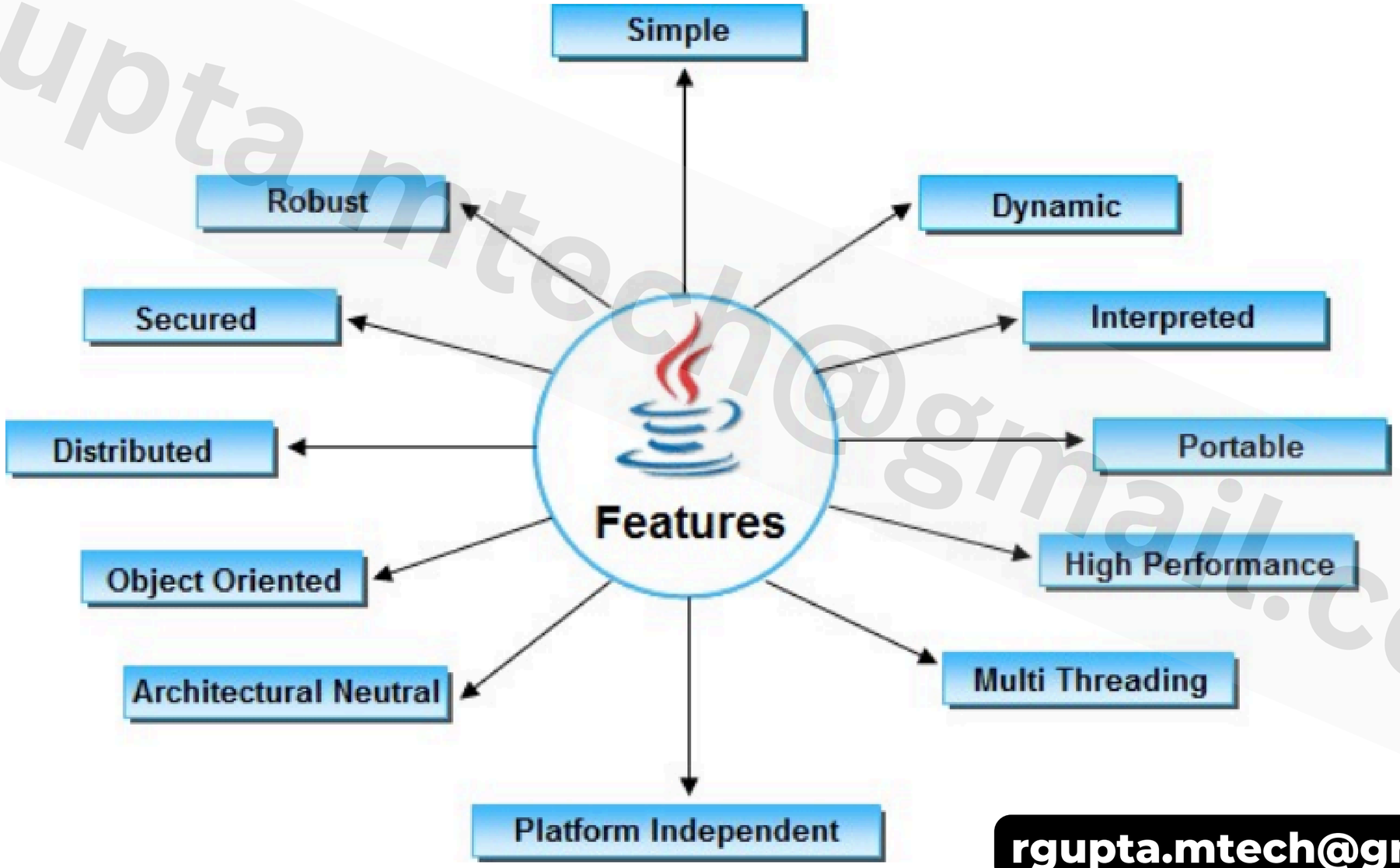
- How Java is different then C++?



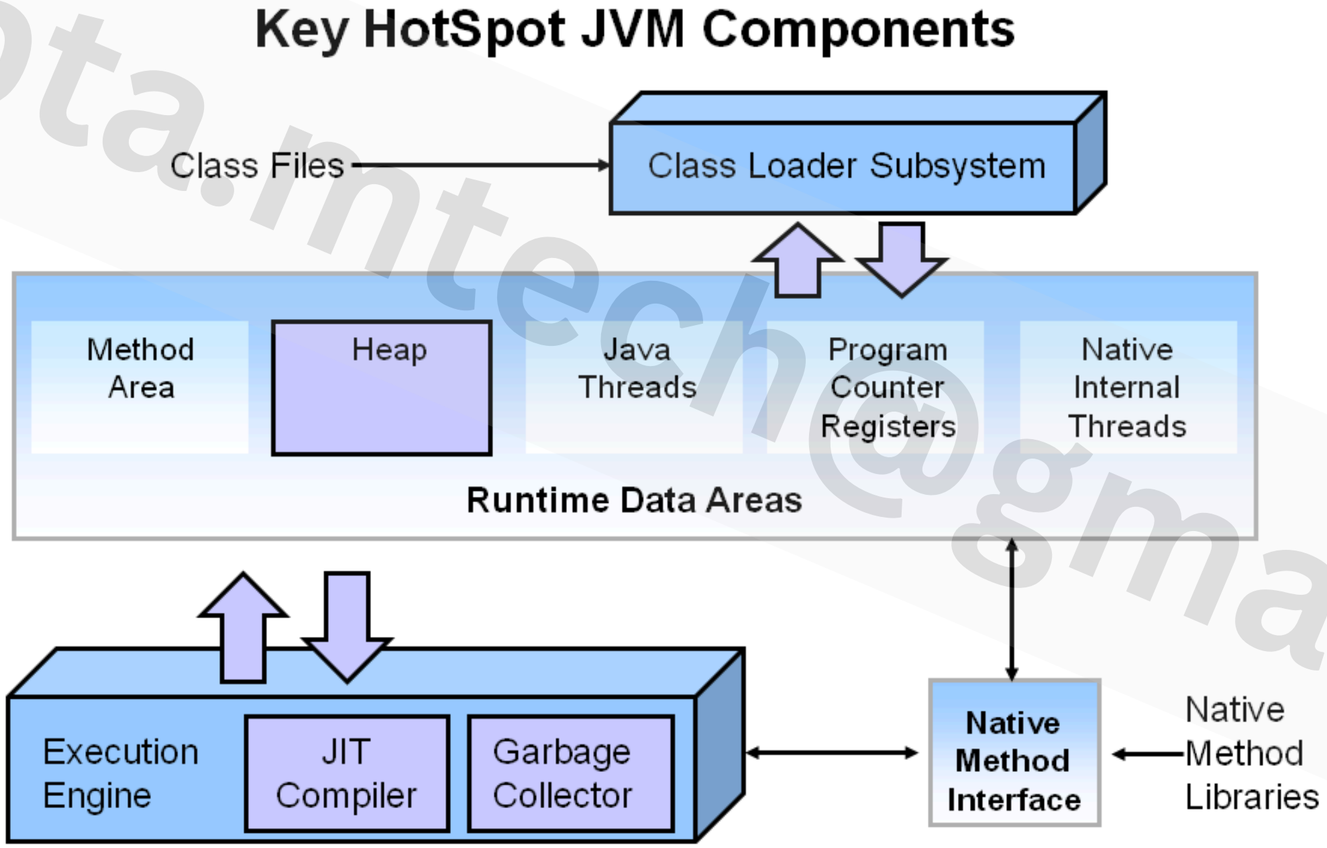
Java Versions



How Sun define Java?



Understanding JVM



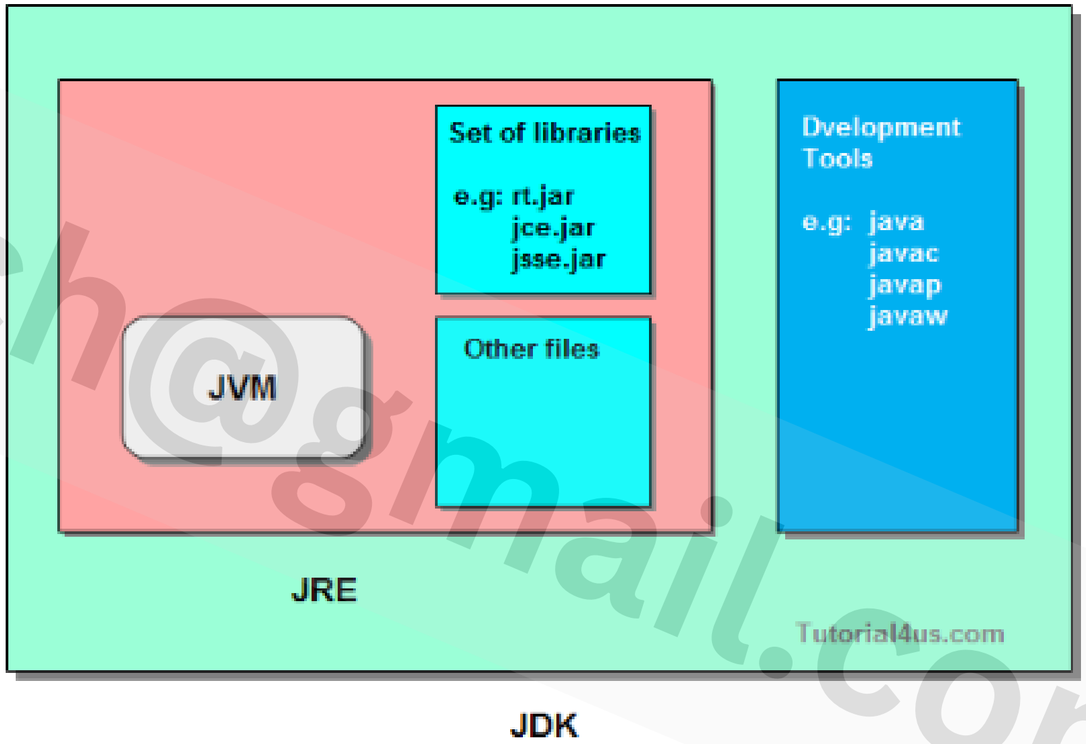
Some common buzzwords

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**



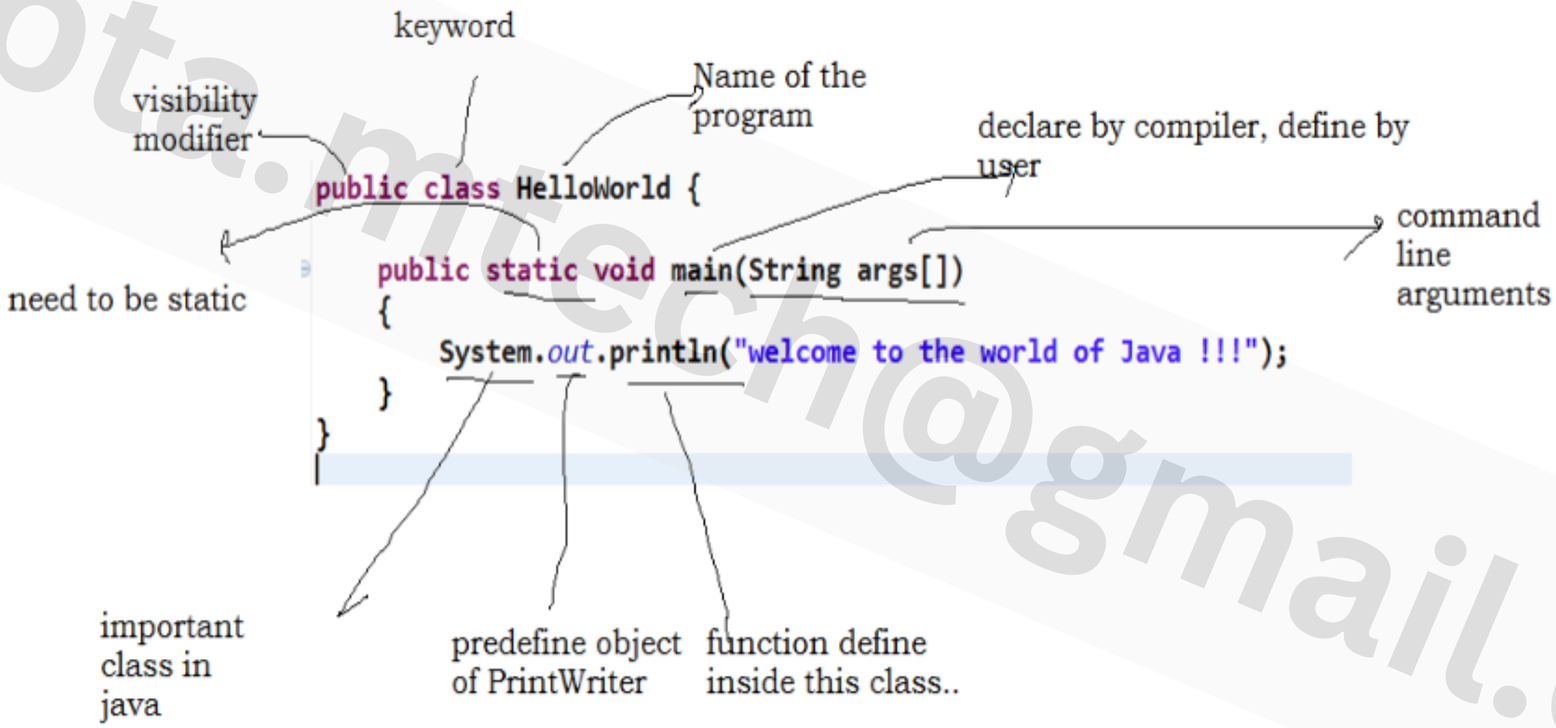
Lab setup and Hello world

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 !!!");
    }
}
```

Analysis of Hello World

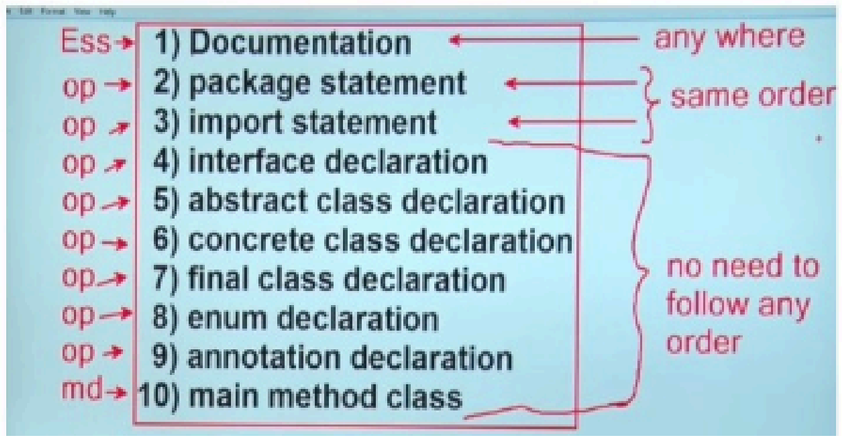


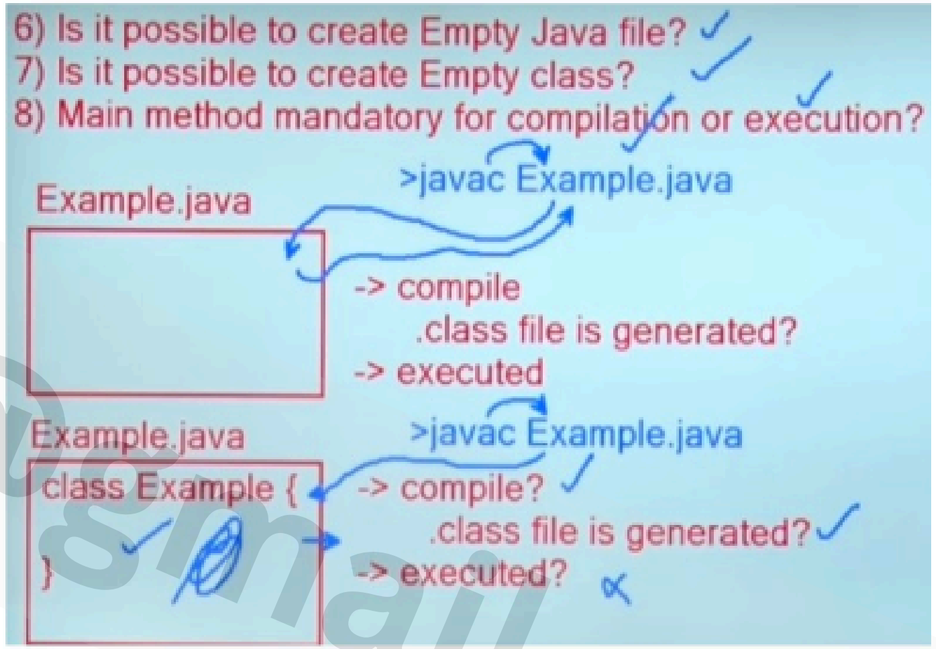
Some basics rules about java classes

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?

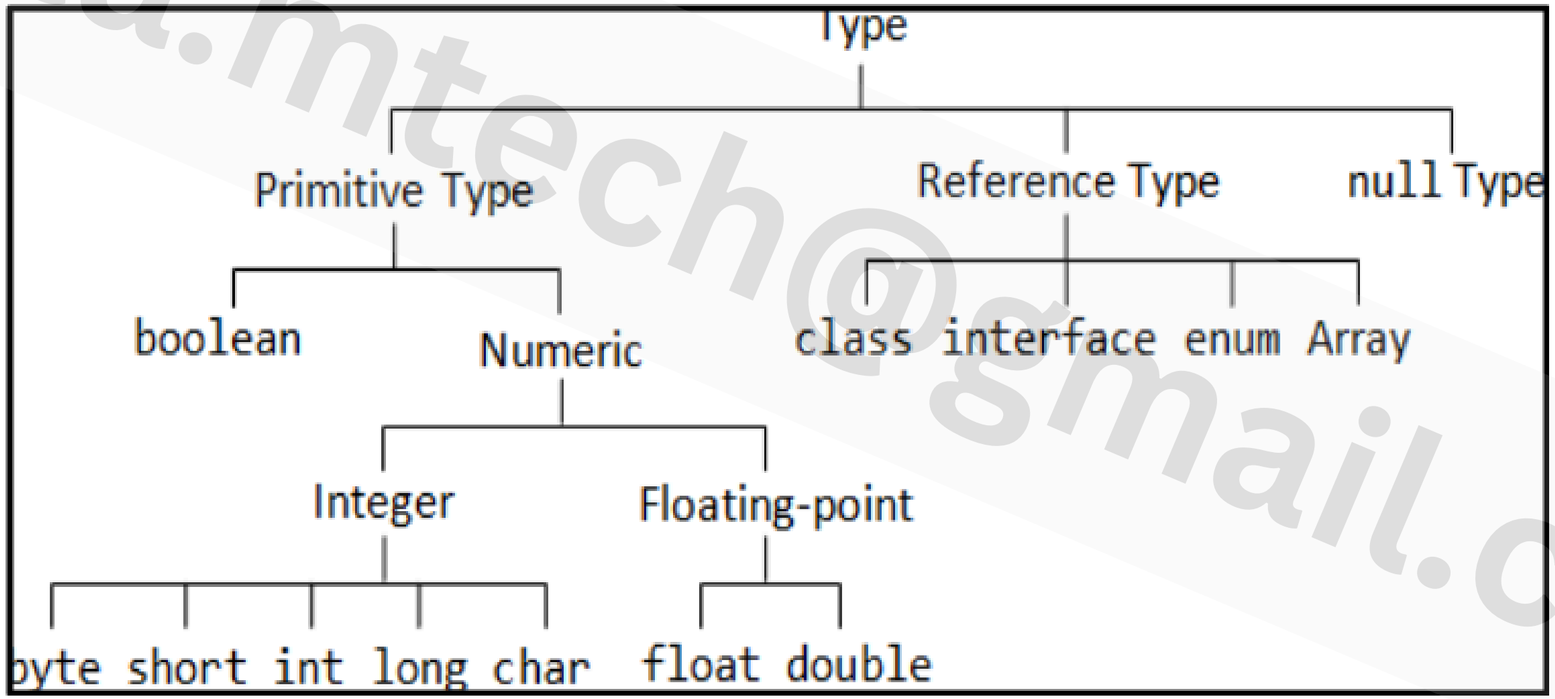




Java Data Type

2 type

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



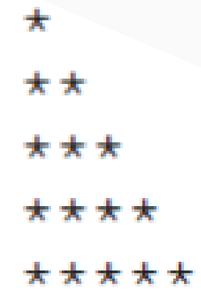
Primitive data type

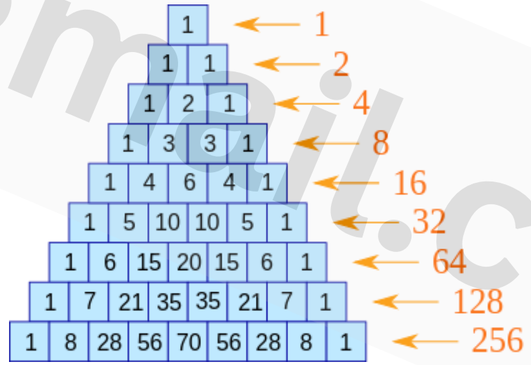
boolean	either true of false
char	16 bit Unicode 1.1
byte	8-bit integer (signed)
short	16-bit integer (signed)
int	32-bit integer (signed)
Long	64-bit integer (singed)
float	32-bit floating point (IEEE 754-1985)
double	64-bit floating point (IEEE 754-1985)

Java uses Unicode to represent characters internally

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





Array are object in java

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

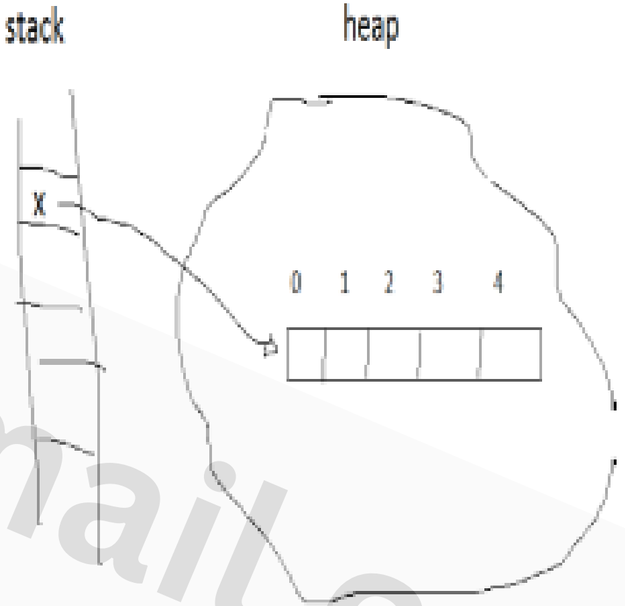
One Dimensional array

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

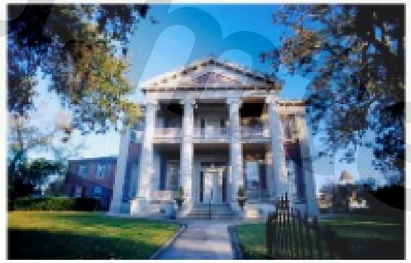


Java Array: its different then C/C++

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

What is Object?

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

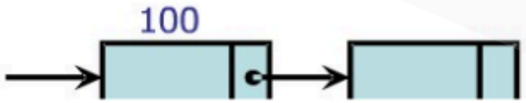
Tom's Car

  - 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

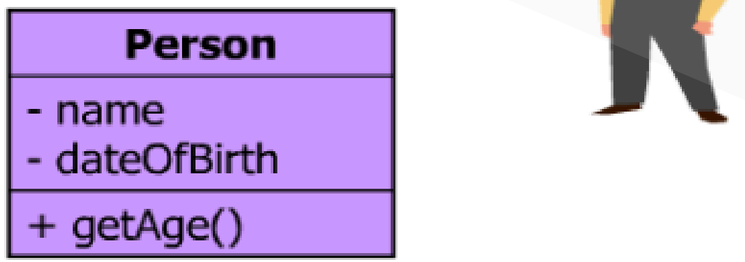
Cookie Cutter

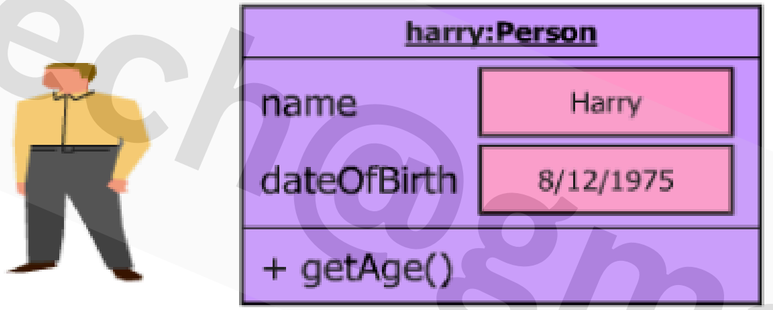
```
public class StudentTest {
  public static void main(String[] args) {
    Student s1 = new Student();
    Student s2 = new Student();
  }
}
```

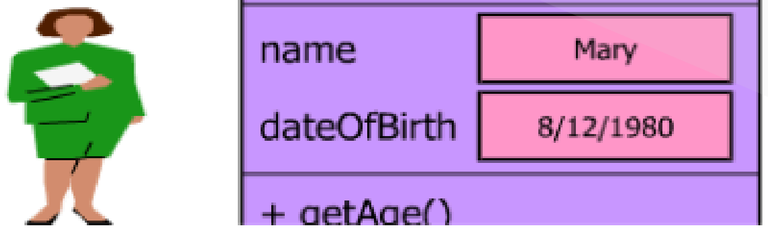
```
- class -

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

class and objects

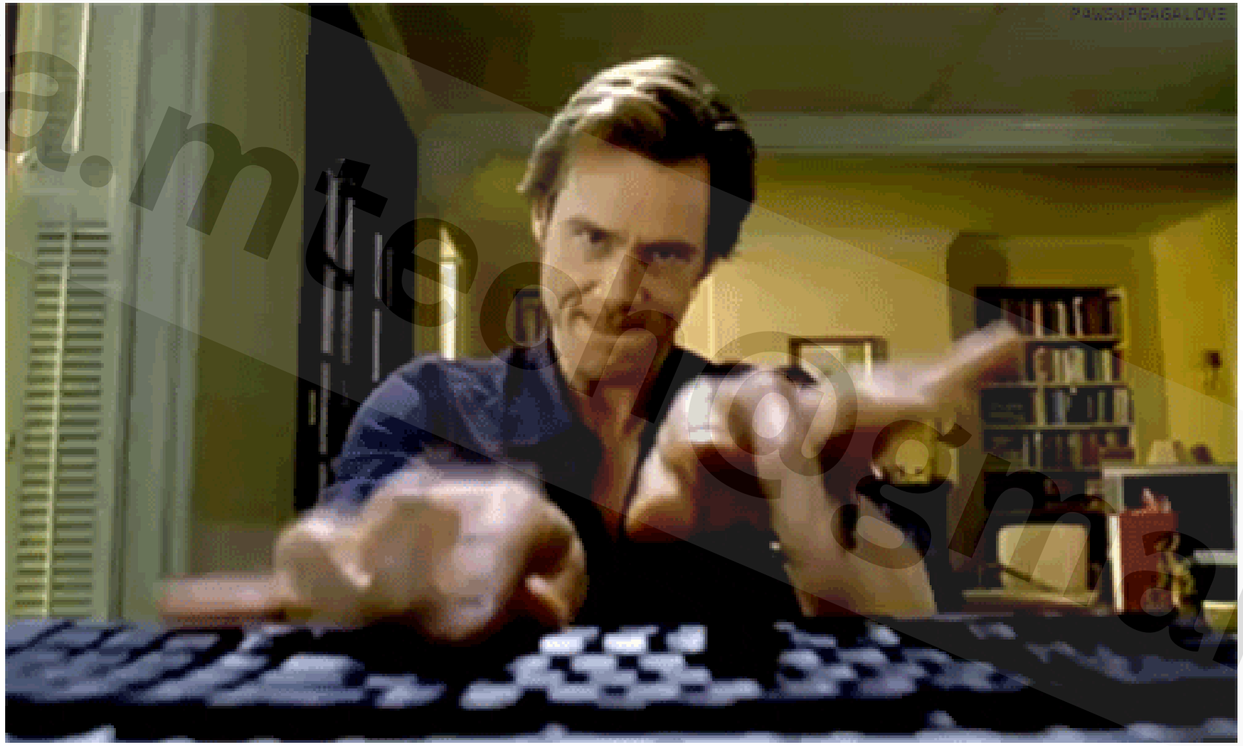
- All the objects share the same attribute names and methods with other objects of the same class
- Each object has its own value for each of the attribute







Programmer are Not typist but thinker



Learning OOPs, SOLID, Design Patterns step by step

Lib & Framework

BTech/MTech

12th class

Design patterns

10th class

SOLID

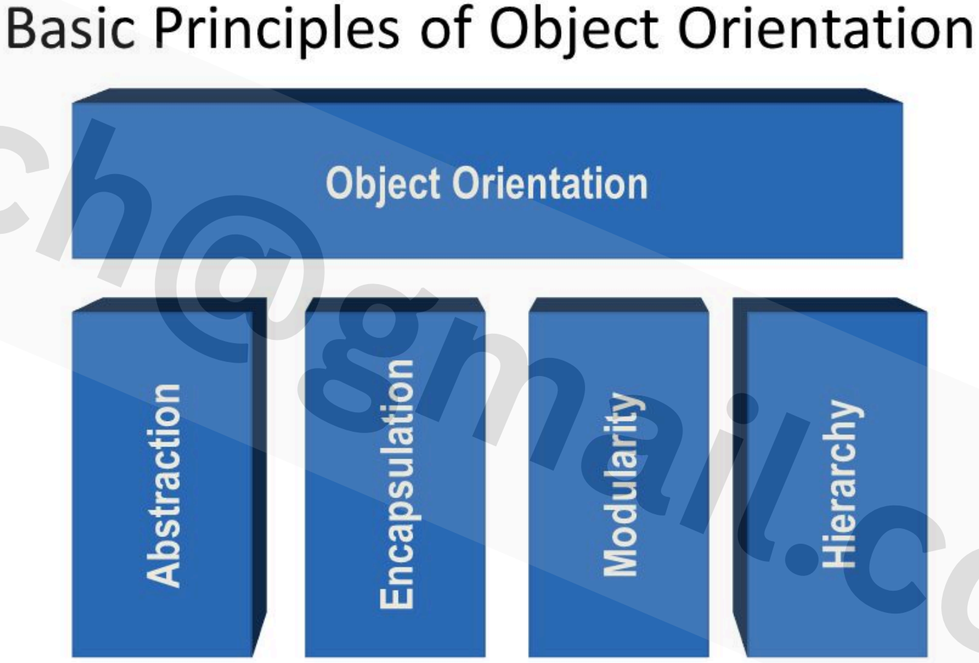
5th class

clear understanding of OOPs concepts

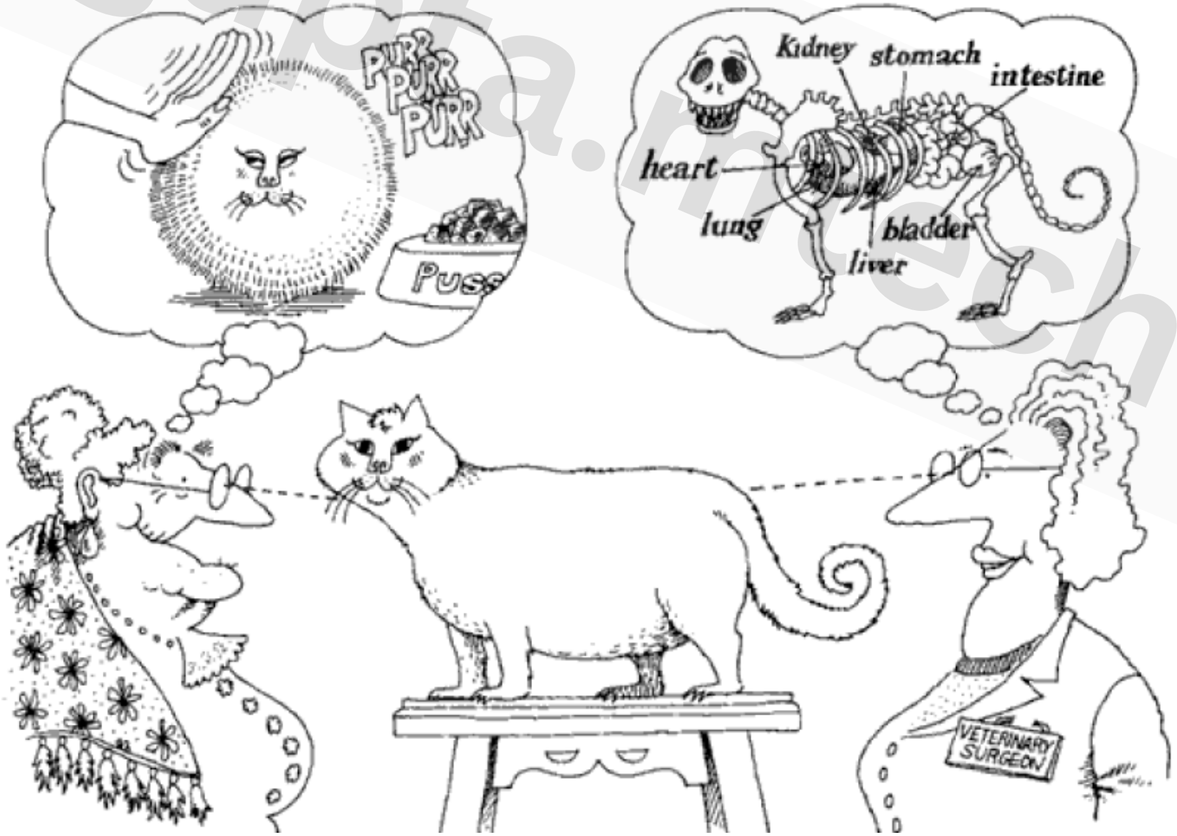
rgupta.mtech@gmail.com

Pillars of object oriented programmin

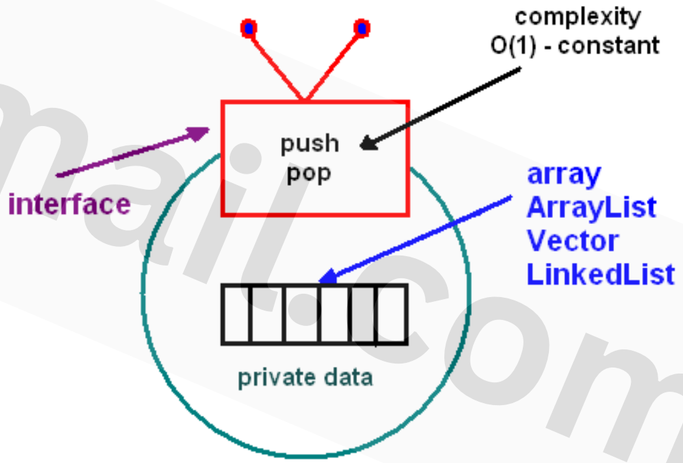
Abstraction
Encapsulation
Modularity
Hierarchy



Abstraction



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

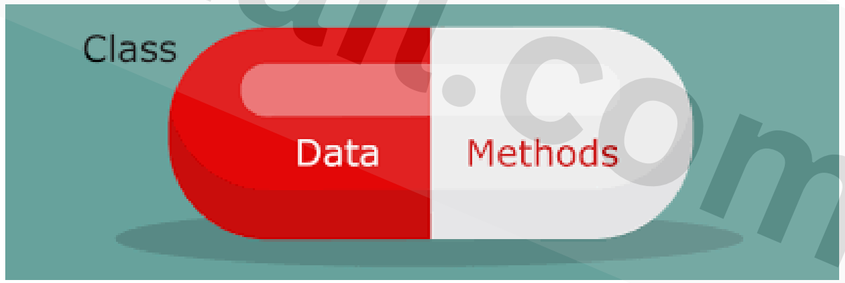


Encapsulation

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



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

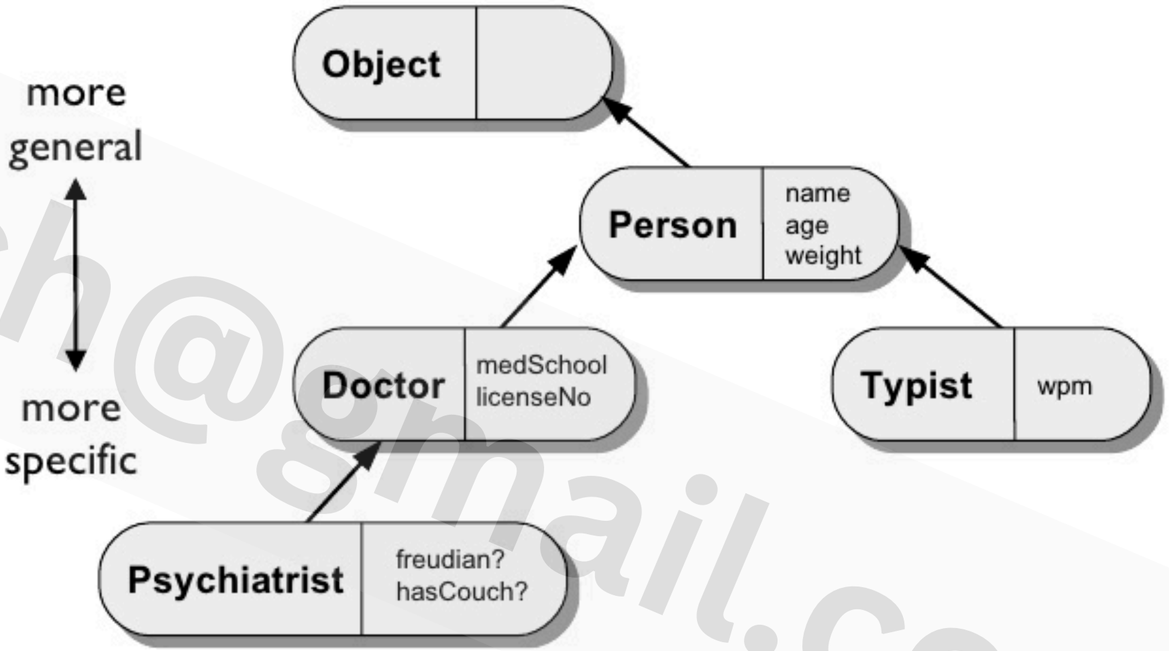
Abstraction and Encapsulation are complementary concepts. Through encapsulation only we are able to enclose the components of the object into a single unit and separate the private and public members. It is through abstraction that only the essential behaviors of the objects are made visible to the outside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

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

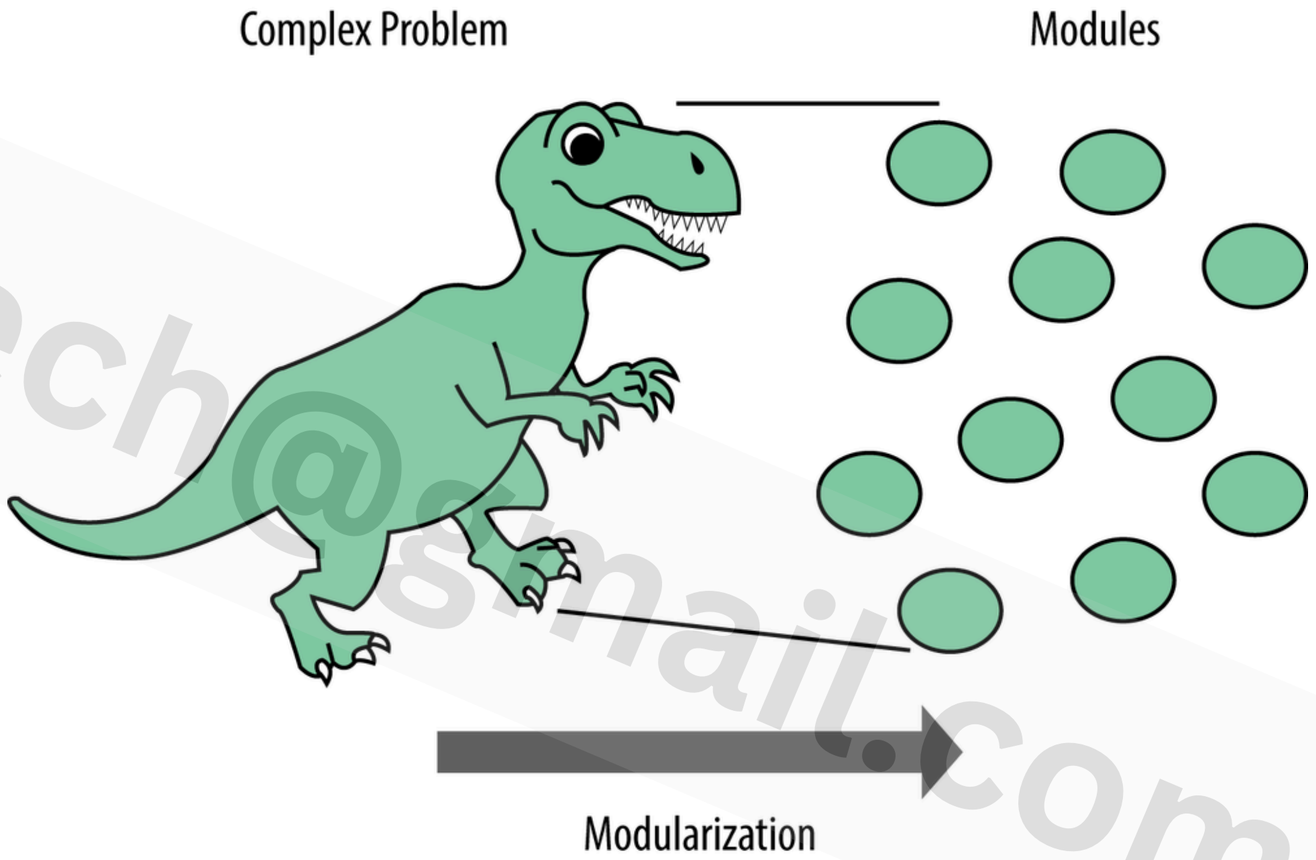
Hierarchy

Hierarchy give rise to inheritance and polymorphism

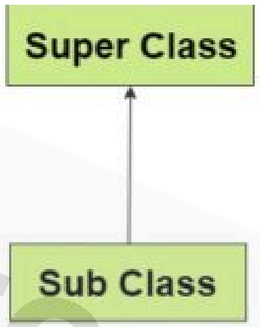


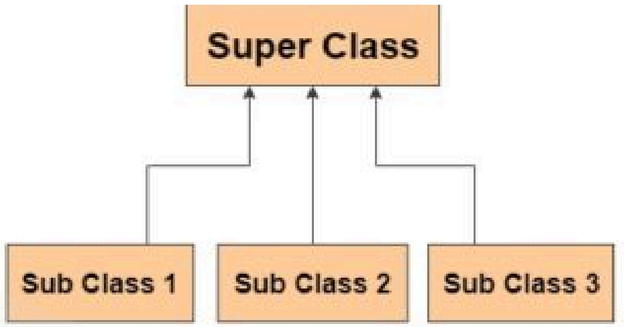
Modularity

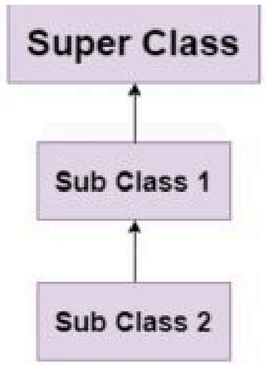
When a class derives from another class. The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods. An inherited class is defined by using the extends keyword



Inheritance

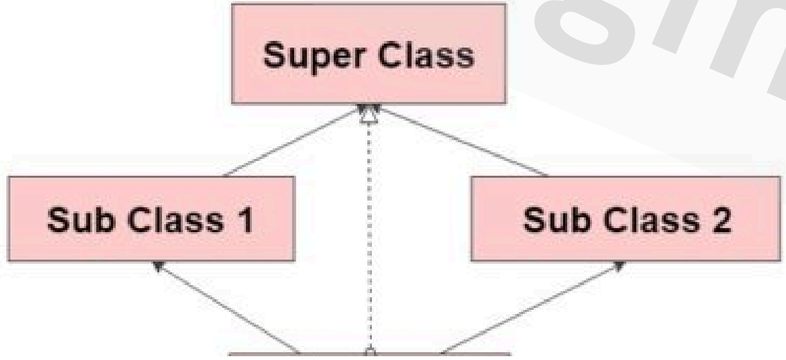




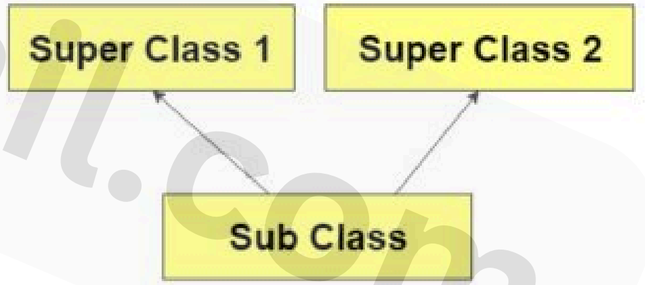


When a class derives from another class. The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods. An inherited class is defined by using the extends keyword

Hybrid Inheritance

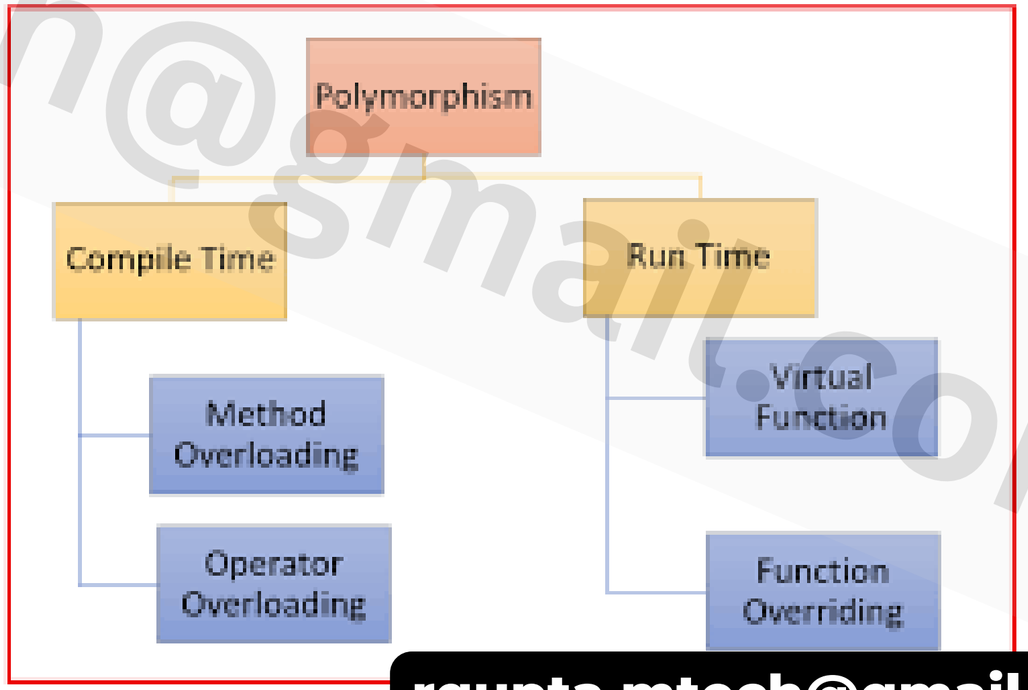


Multiple Inhertance



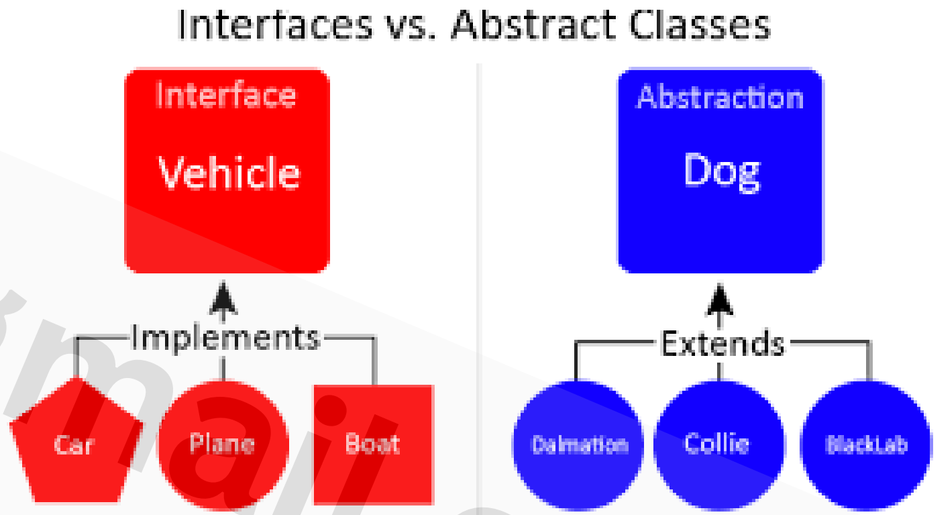
Polymorphism

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.



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	✓	✗

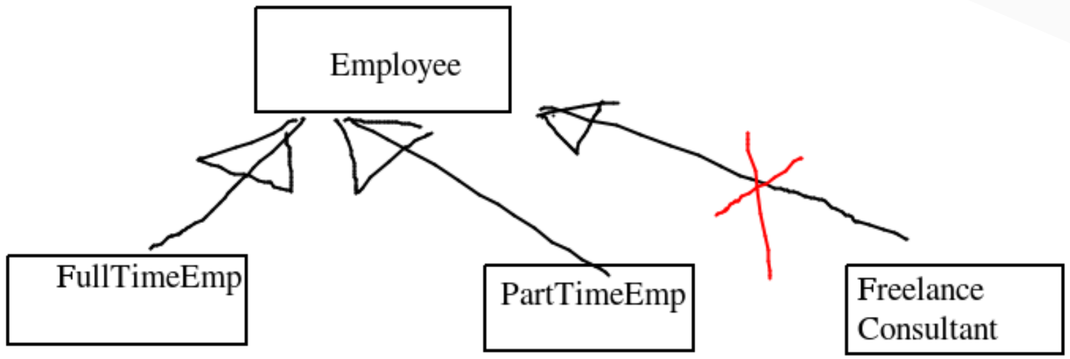


Interface vs Abstract class

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

What an object can do?

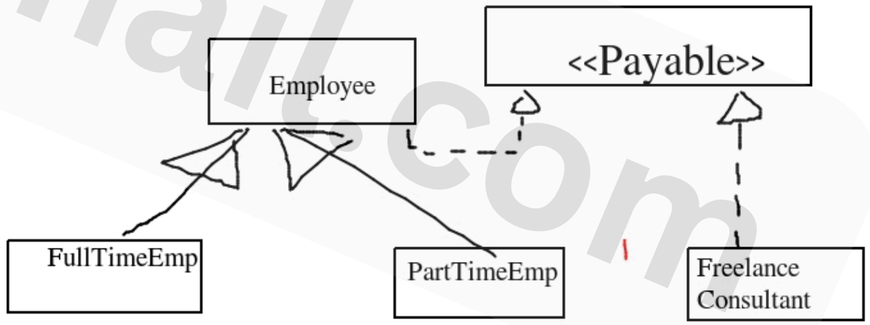
Standardization of a behavior



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

Session 2:
Basic OOPs, Inheritance, Overriding, Overloading, Polymorphism Advance Object Orientation

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) {
            //........
        }
}
```

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

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) {
            //
```

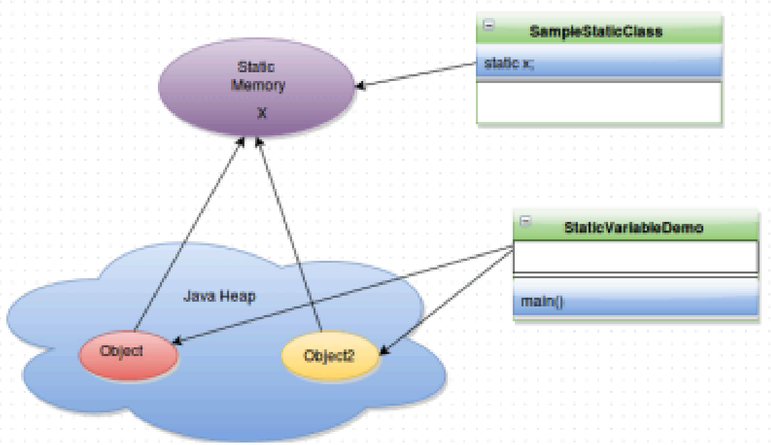
Calling one constructor from another ?

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  ...");
   }
}
```

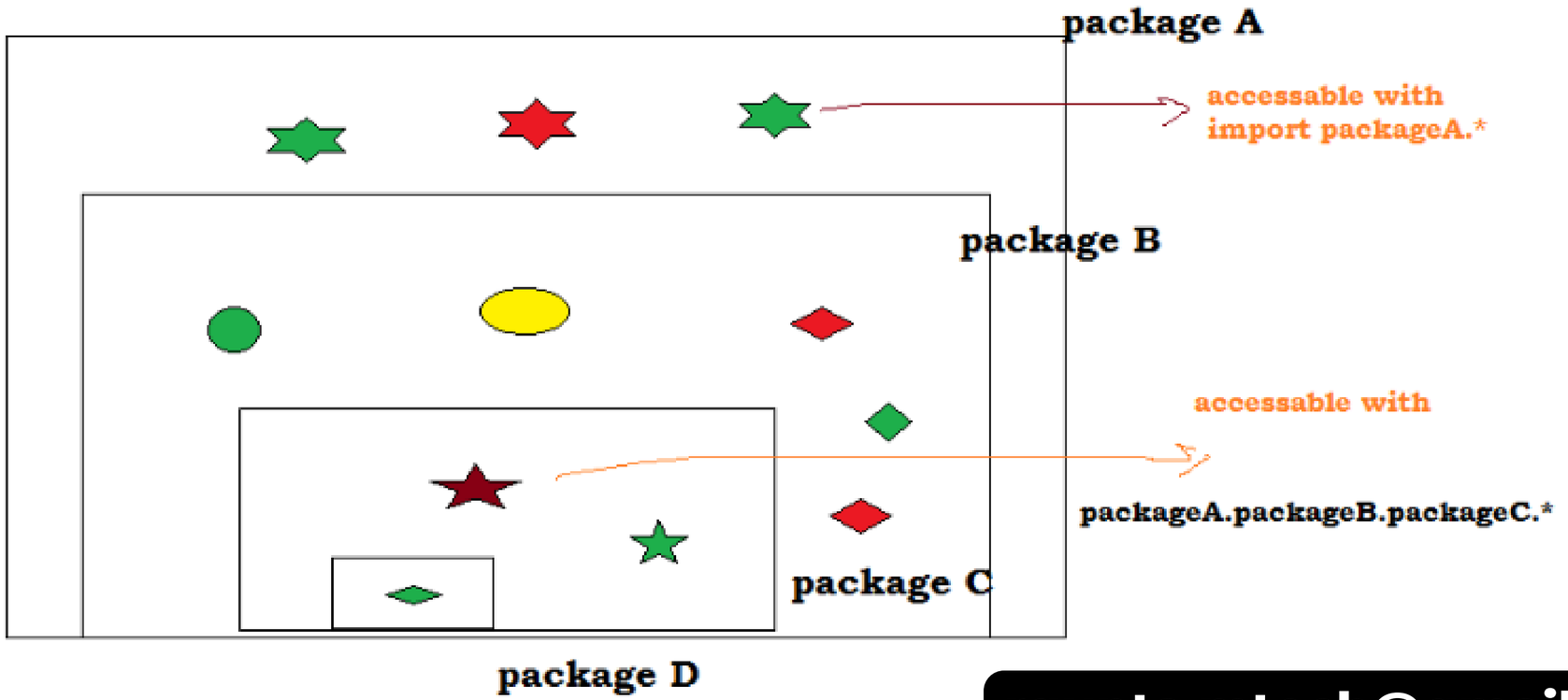
```
class Account{

   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

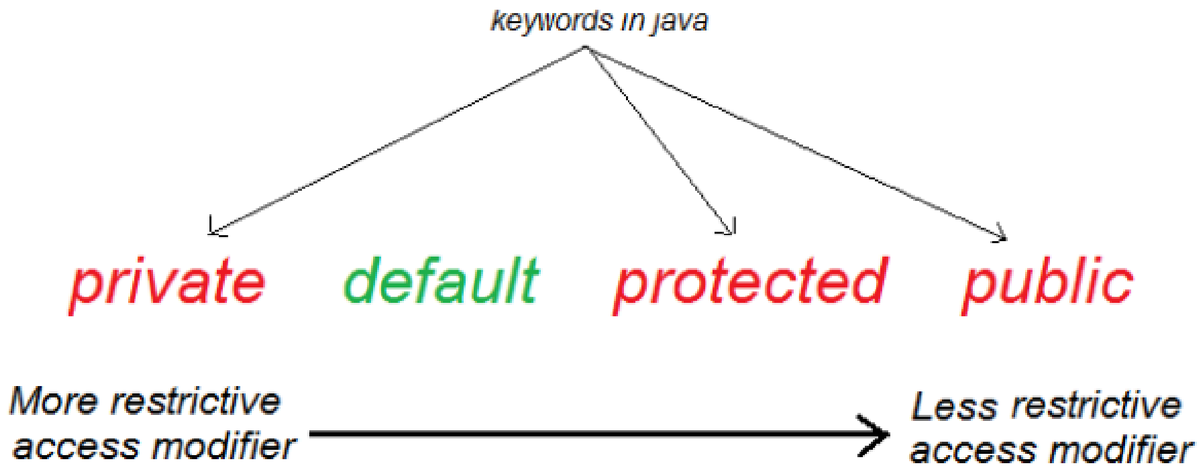
- Packages are Java's way of grouping a number of related classes and/or interfaces together into a single unit.
- Packages act as "containers" for classes.



rgupta.mtech@gmail.com

visibility modifiers

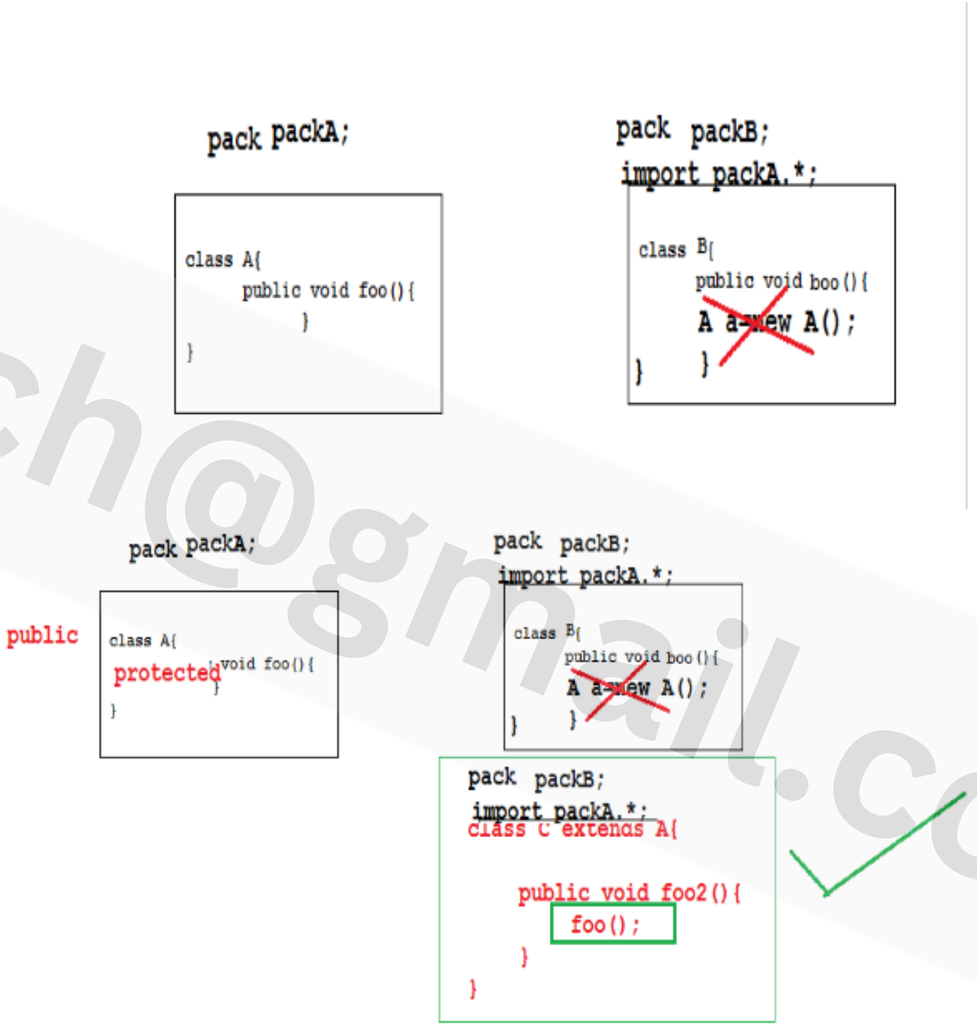
- For instance variable and methods
  - Public
  - Protected
  - Default (package level)
  - Private
- For classes
  - Public and default



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



Want to accept parameter from user?

java.util.Scanner (Java 1.5)

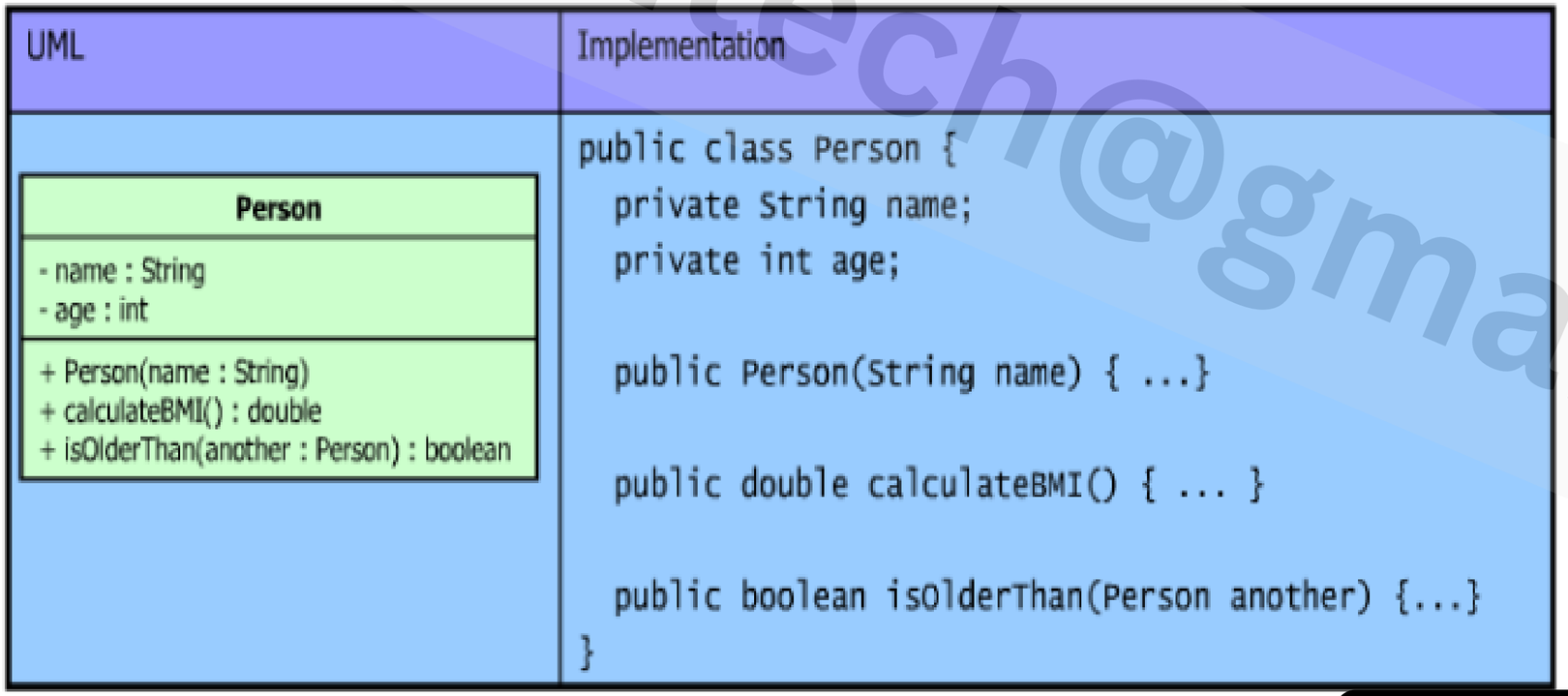
```
Scanner stdin = Scanner.create(System.in);  int n = stdin.nextInt();
String s = stdin.next();
```

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


rgupta.mtech@gmail.com

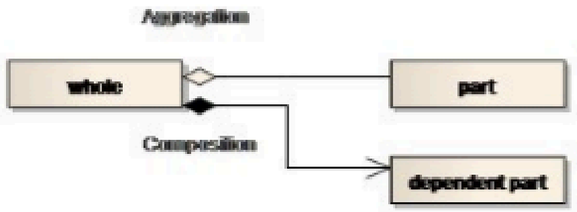
UML diagram Basics

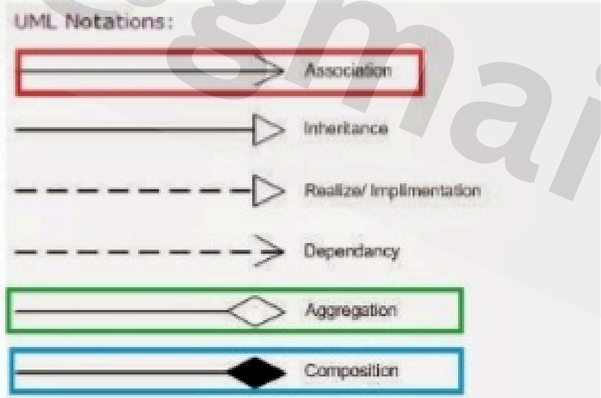
- **UML 2.0 aka modeling language has 12 type of diagrams**
- **Most important once are class diagram, use case diagram and sequence diagram.**
- **You should know how to read it correctly**



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



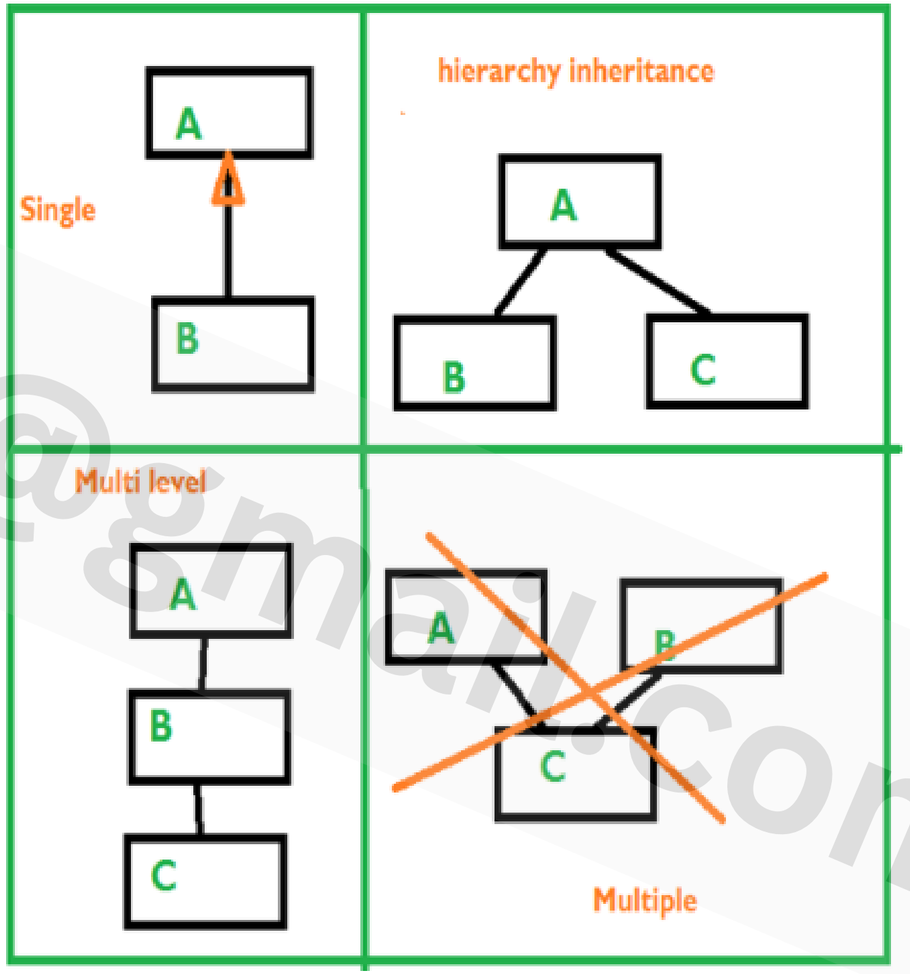


Inheritance

- **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.**

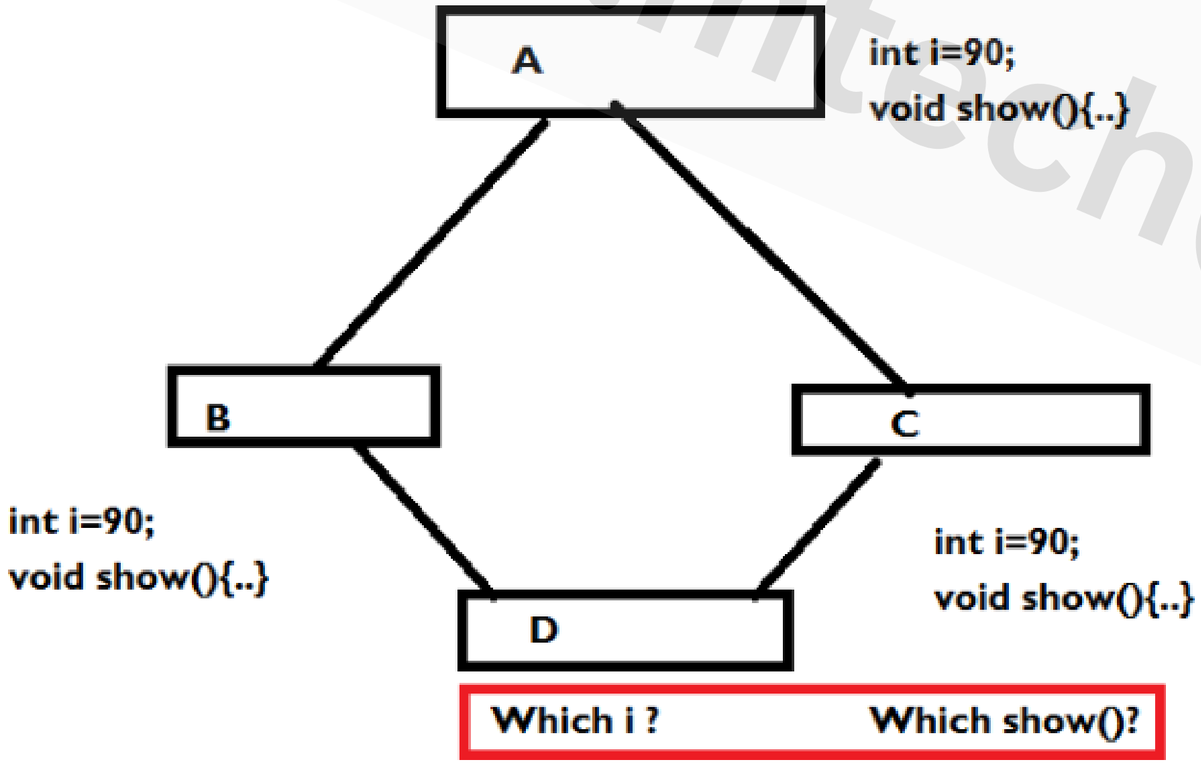
- **Code reusability.**

- **Subclass and Super class concept**



Diamond Problem?

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



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");
    }

}
```

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...**

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) { ...}
}
```

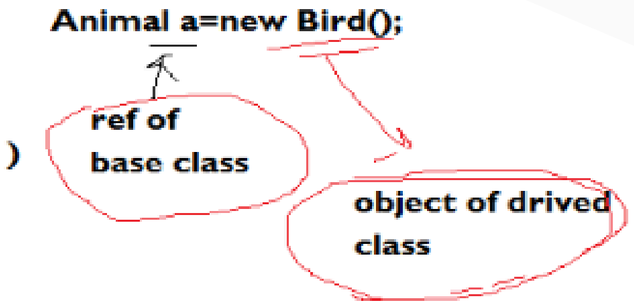
Polymorphism

- 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.

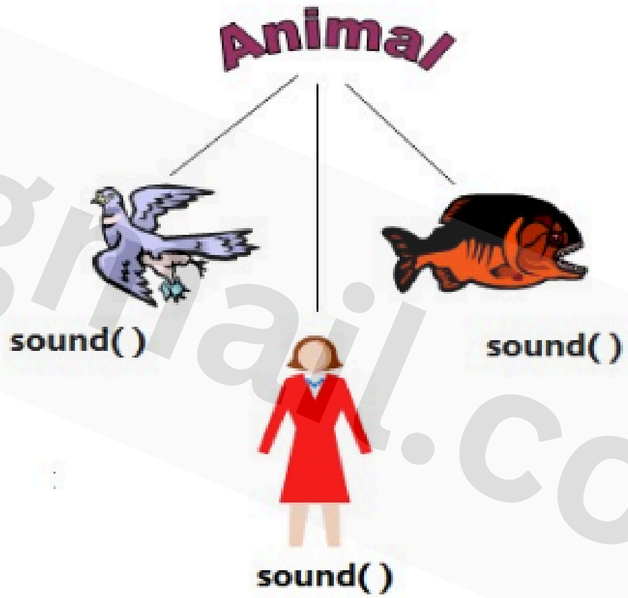
rgupta.mtech@gmail.com

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



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.....");
    }
}
```

Day 2:
Abstract class, Interface, Inner classes
String, Exception handling, IO

rgupta.mtech@gmail.com

Day-2
Session -1
Interface & Abstract class

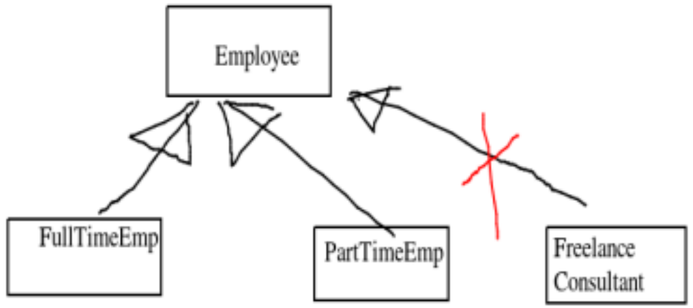
rgupta.mtech@gmail.com

Interface vs Abstract class

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

What an object can do?

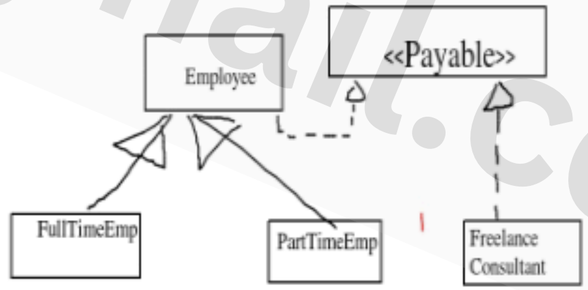
Standardization of a behavior



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

Need of abstract class?

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

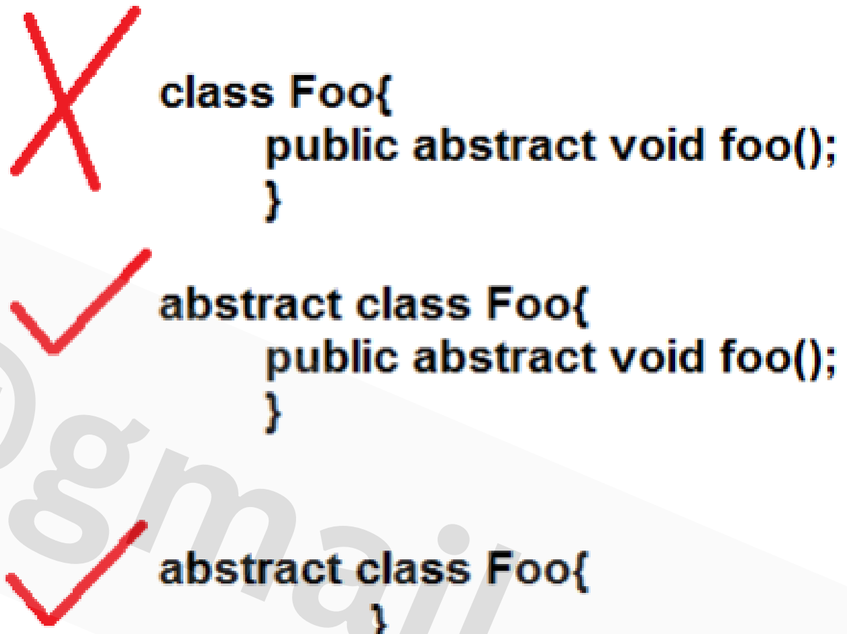
- **Method sound( ) in Animal class should be abstract means incomplete**

- **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.....");
    }
}
```

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



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

Example Abstract class

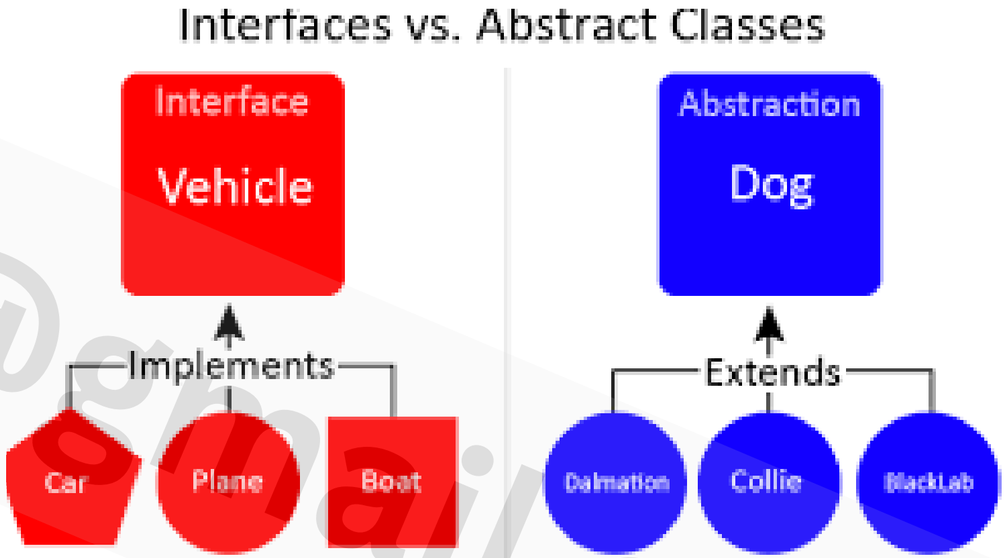
```
public abstract class Account  {
    public void deposit (double amount) {
        System.out.println("depositing " + amount);
    }

    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
    }
}
```

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	✓	✗



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");
     }
}
```

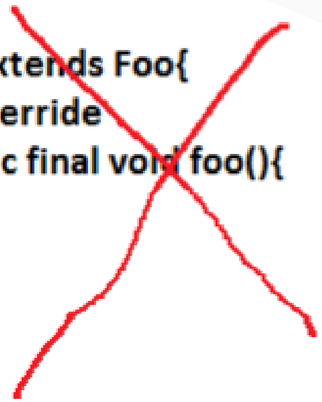
```
package examStuff;
import cart.*;

class Tea extends beverage{

}
```


rgupta.mtech@gmail.com

Final Method

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

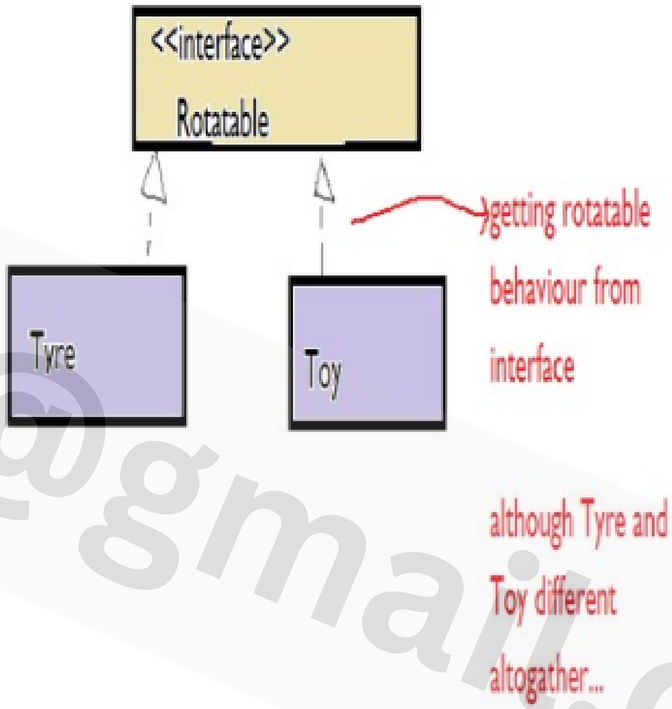
```
class Foo{

        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**



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....

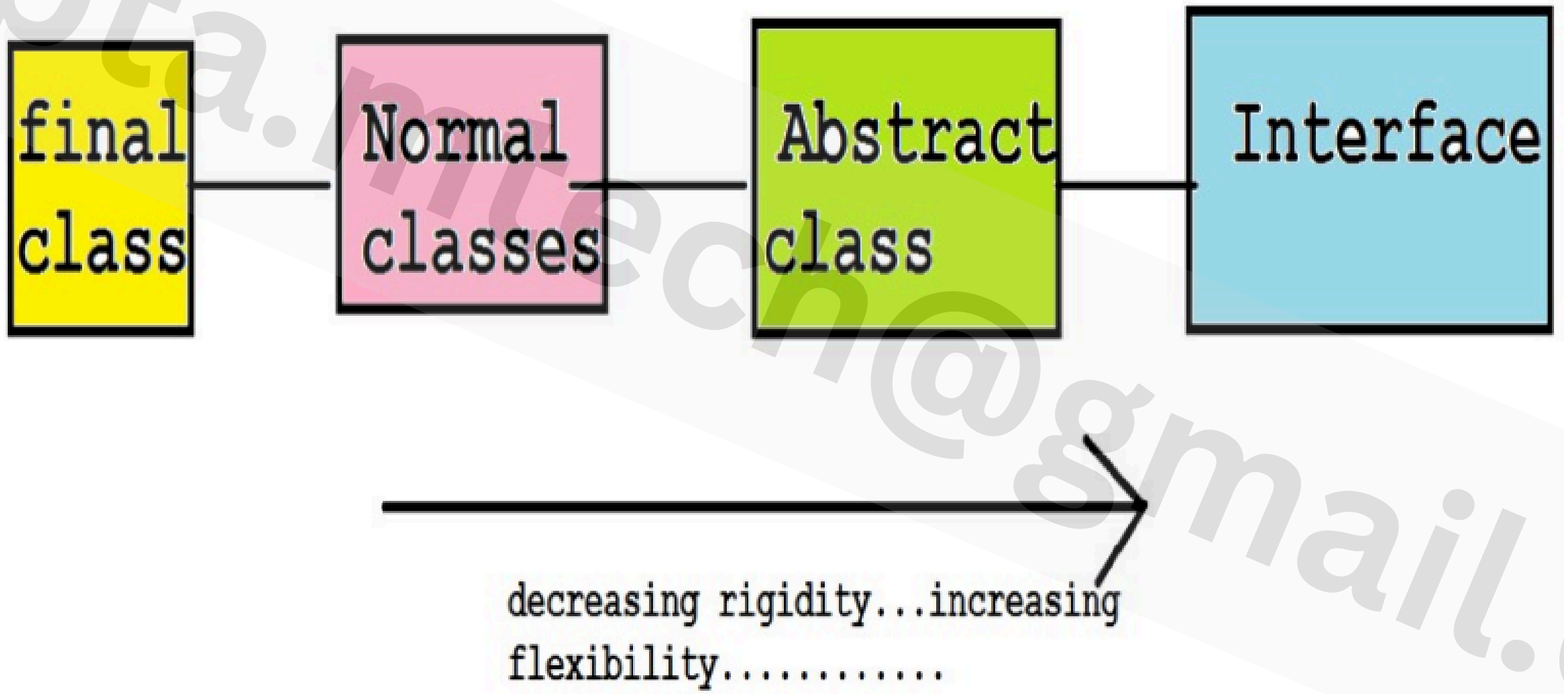
```
class Tyre implements Bouncable{

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

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



Type of relationship bw objects

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

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

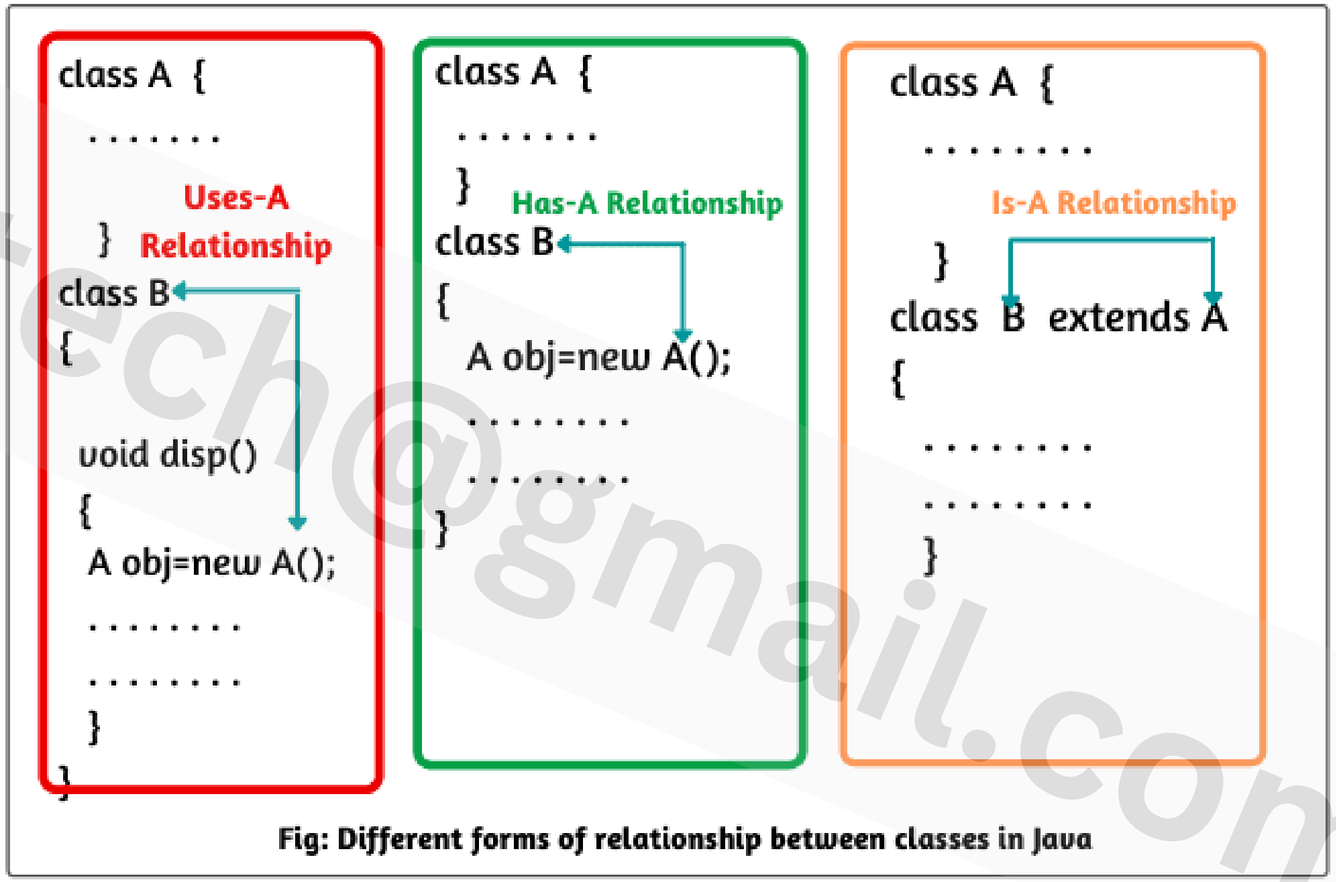
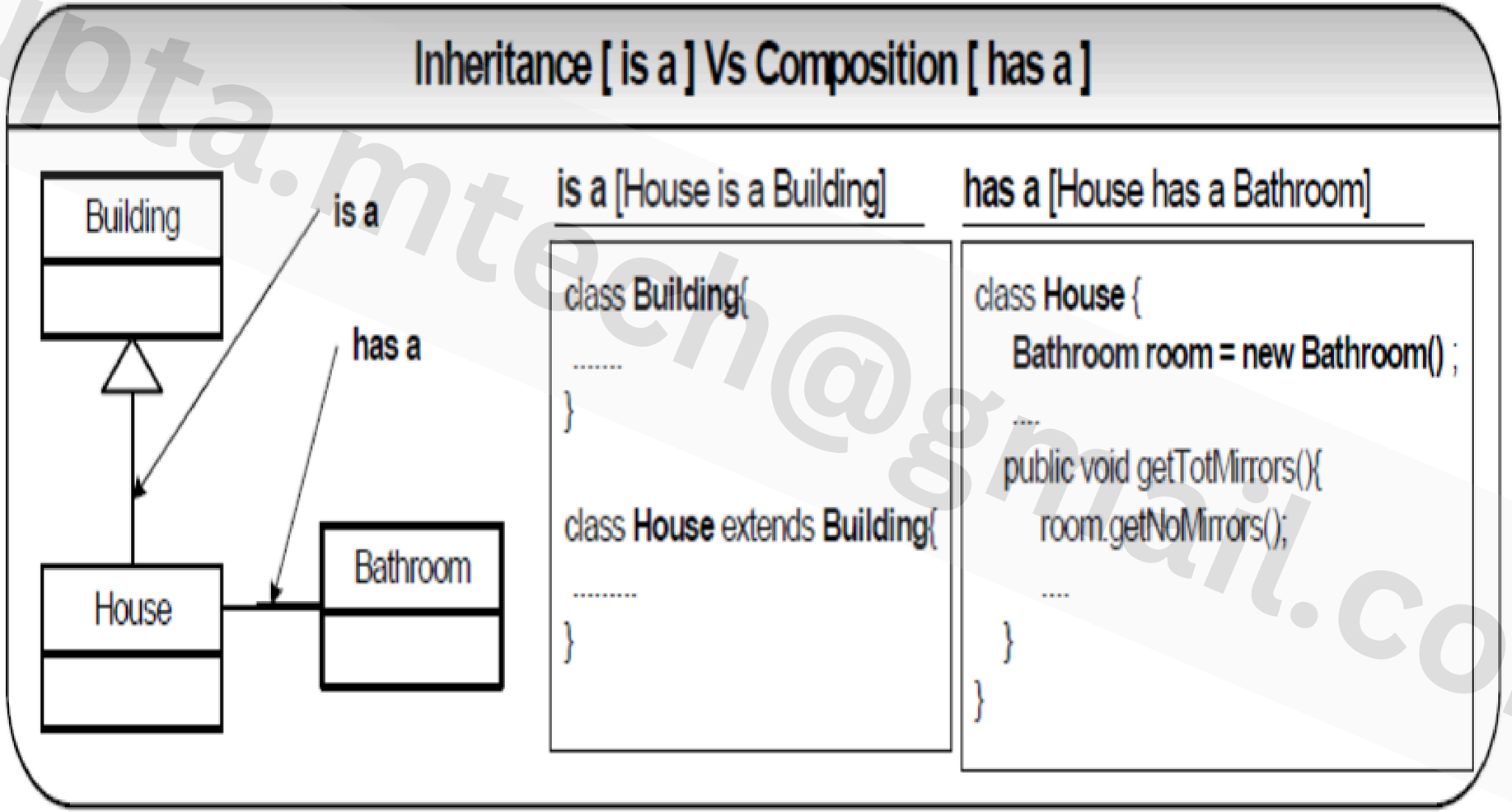


Fig: Different forms of relationship between classes in Java

Inheritance vs Composition



Day-2
Session 2
String, Wrapper classes,Java 5 features
Inner classes

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

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

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))
.....
.....

String vs StringBuilder vs StringBuffer

	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

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.

```
//Inefficient version using immutable String
String output = "Some text"
Int count = 100;
for(int i =0; i<count; i++) {
    output += i;
}
return output;
```

The above code would build 99 new String objects, of which 98 would be thrown away immediately. Creating new objects is not efficient.

StringBuffer / StringBuilder (added in J2SE 5.0)

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.

```
//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 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.

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)
```

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();
```

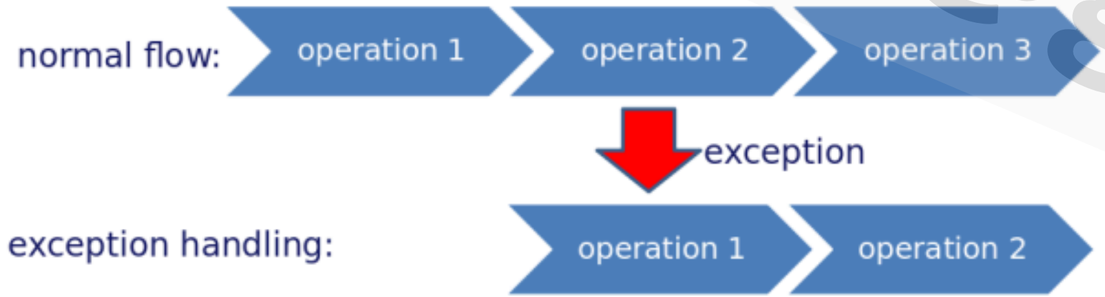
Day-2
Session 2:
Java Exception Handling, IO

rgupta.mtech@gmail.com

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...

Type of exceptions

1. Unchecked Exception Also called Runtime Exceptions

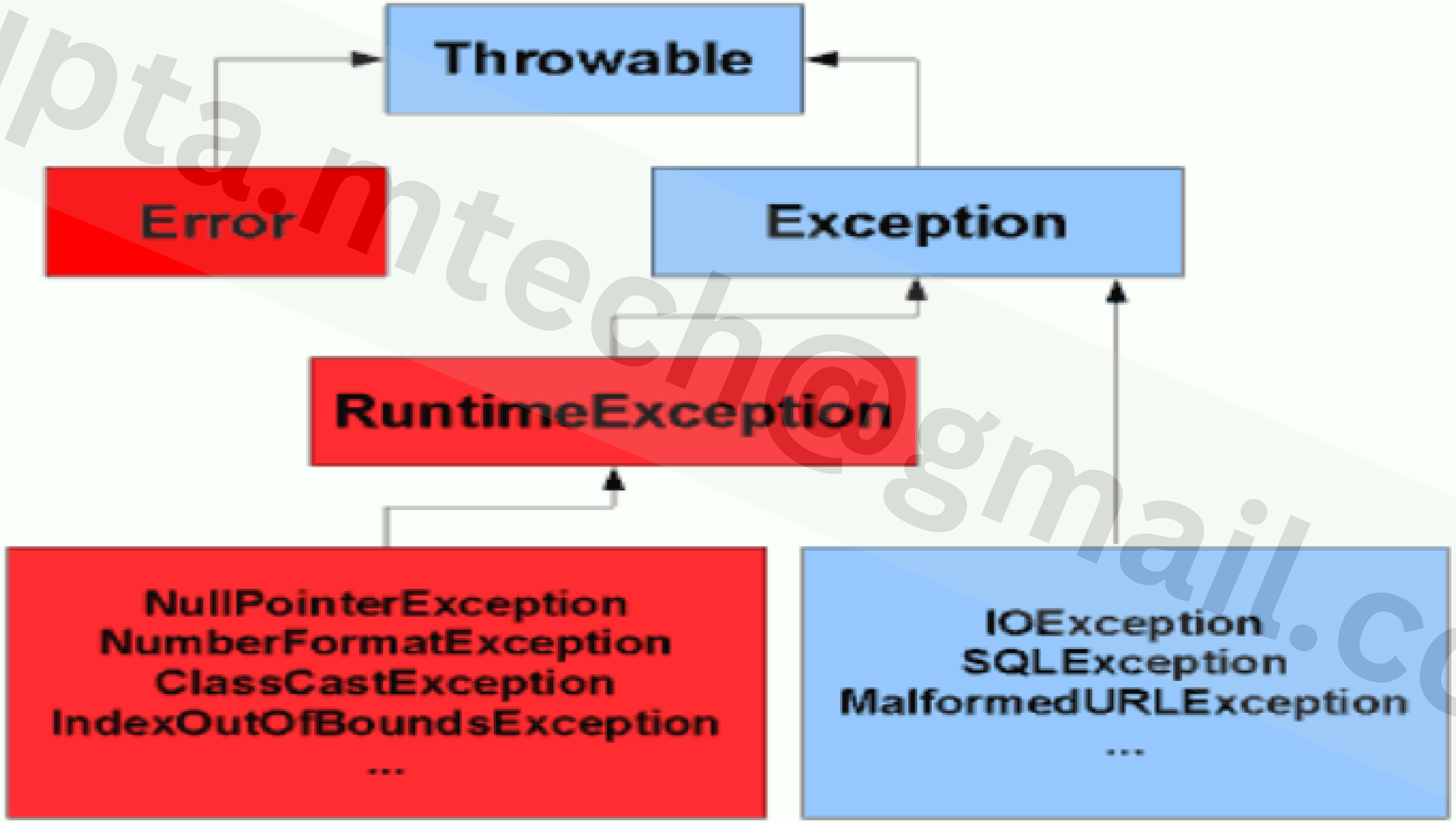
2. Checked Exception
Also called Compile time Exceptions

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

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

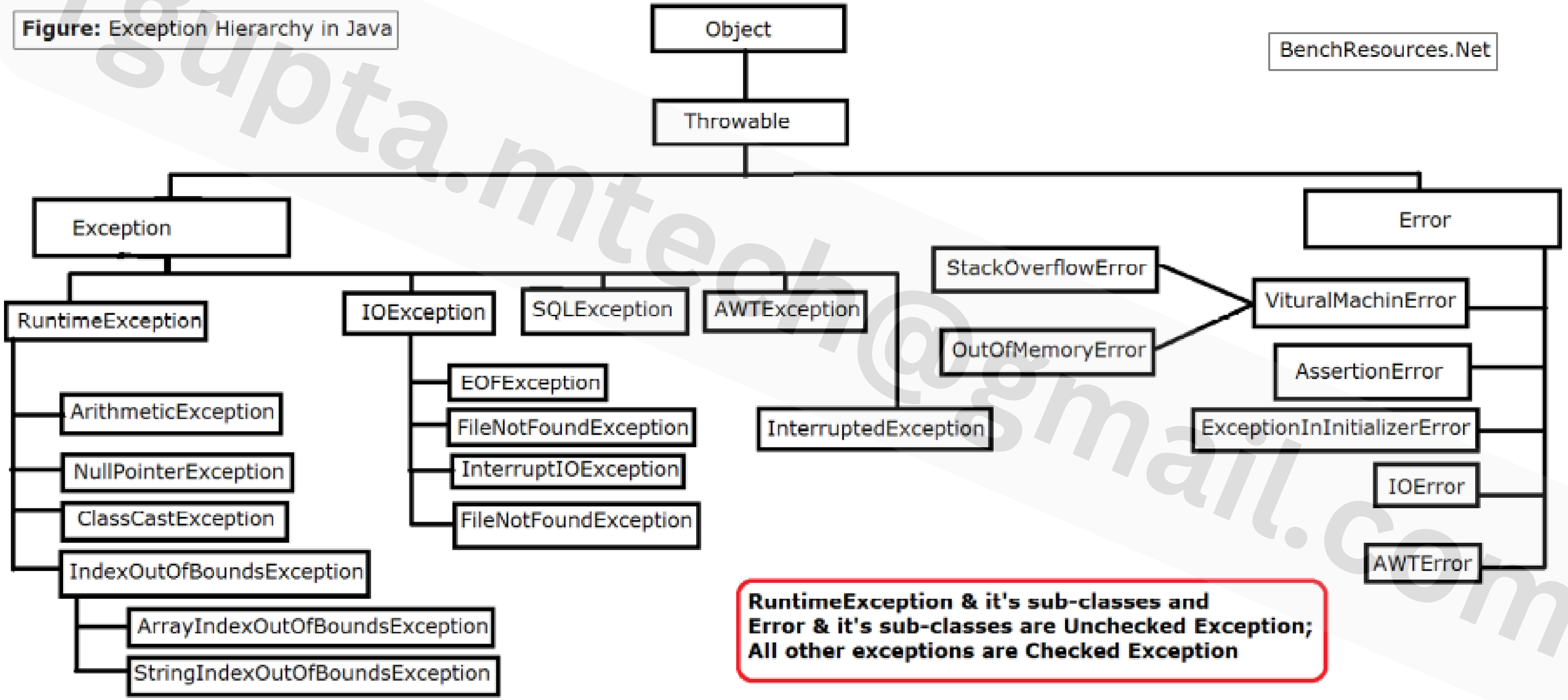
All exceptions happens at run time in programming and also in real life.....
for checked ex, we need to tell java we know about those problems for example
readLine() throws IOException

Exception Hierarchy



Exception Hierarchy

Figure: Exception Hierarchy in Java



BenchResources.Net

RuntimeException & it's sub-classes and Error & it's sub-classes are Unchecked Exception; All other exceptions are Checked Exception

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

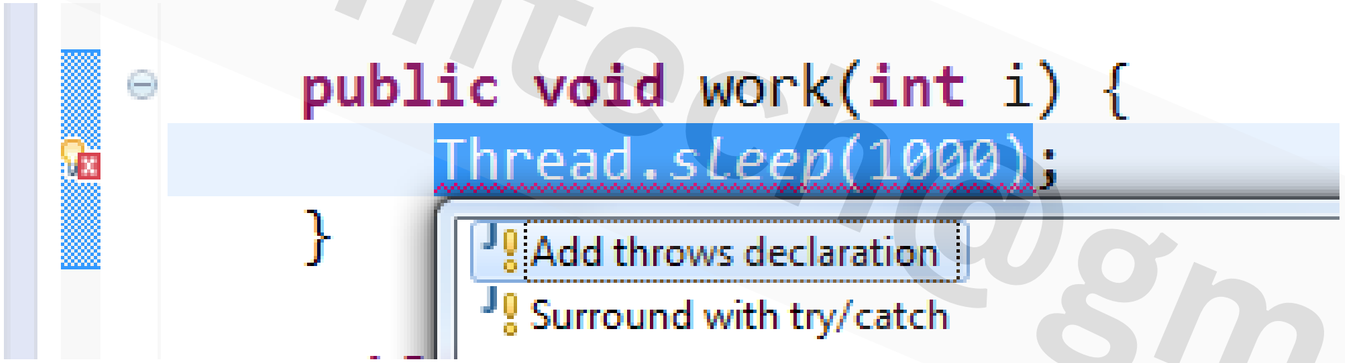
- **A finally clause is executed even if a return statement is executed in the try or catch clauses.**
- **An uncaught or nested exception still exits via the finally clause.**
- **Typical usage is to free system resources before returning, even after throwing an exception (close files, network links)**

```
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

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);
}
```

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");

}
```

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() {  }
}
```

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**

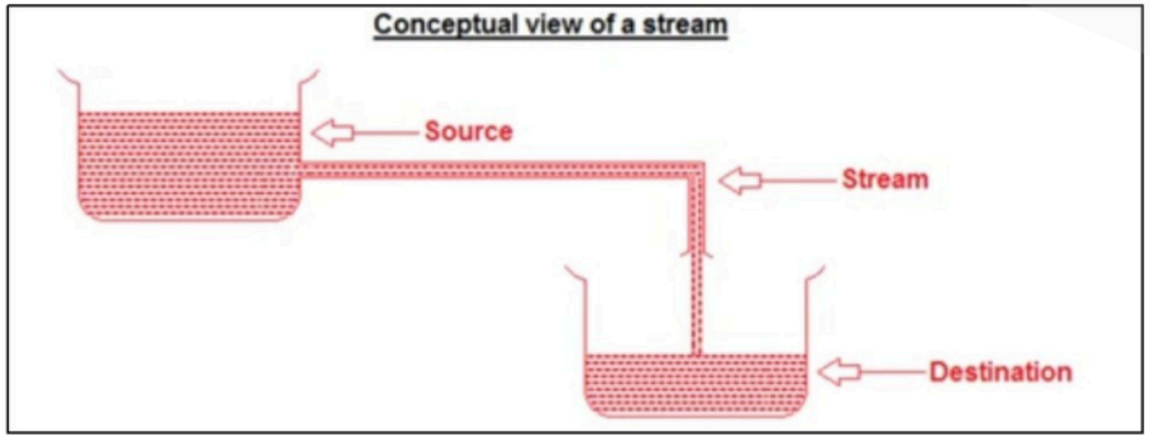
The stack trace can be printed by calling printStackTrace()

```
// 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

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



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.

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.
```

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:

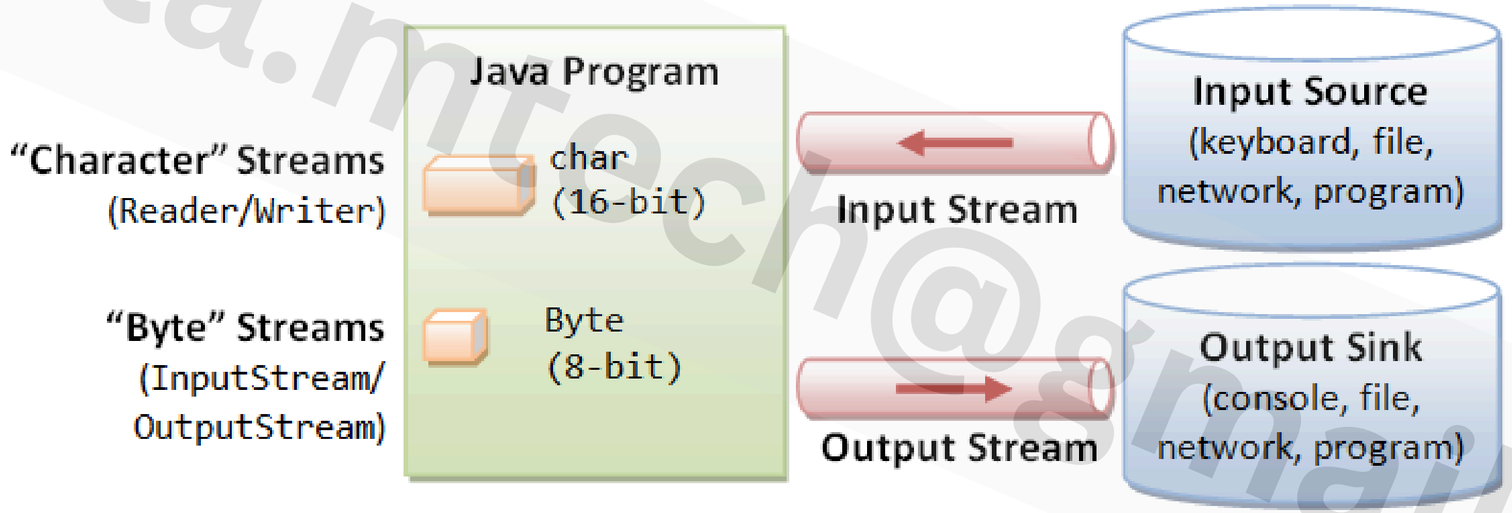
```
public boolean accept(File dirName, String fileName)
```

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)



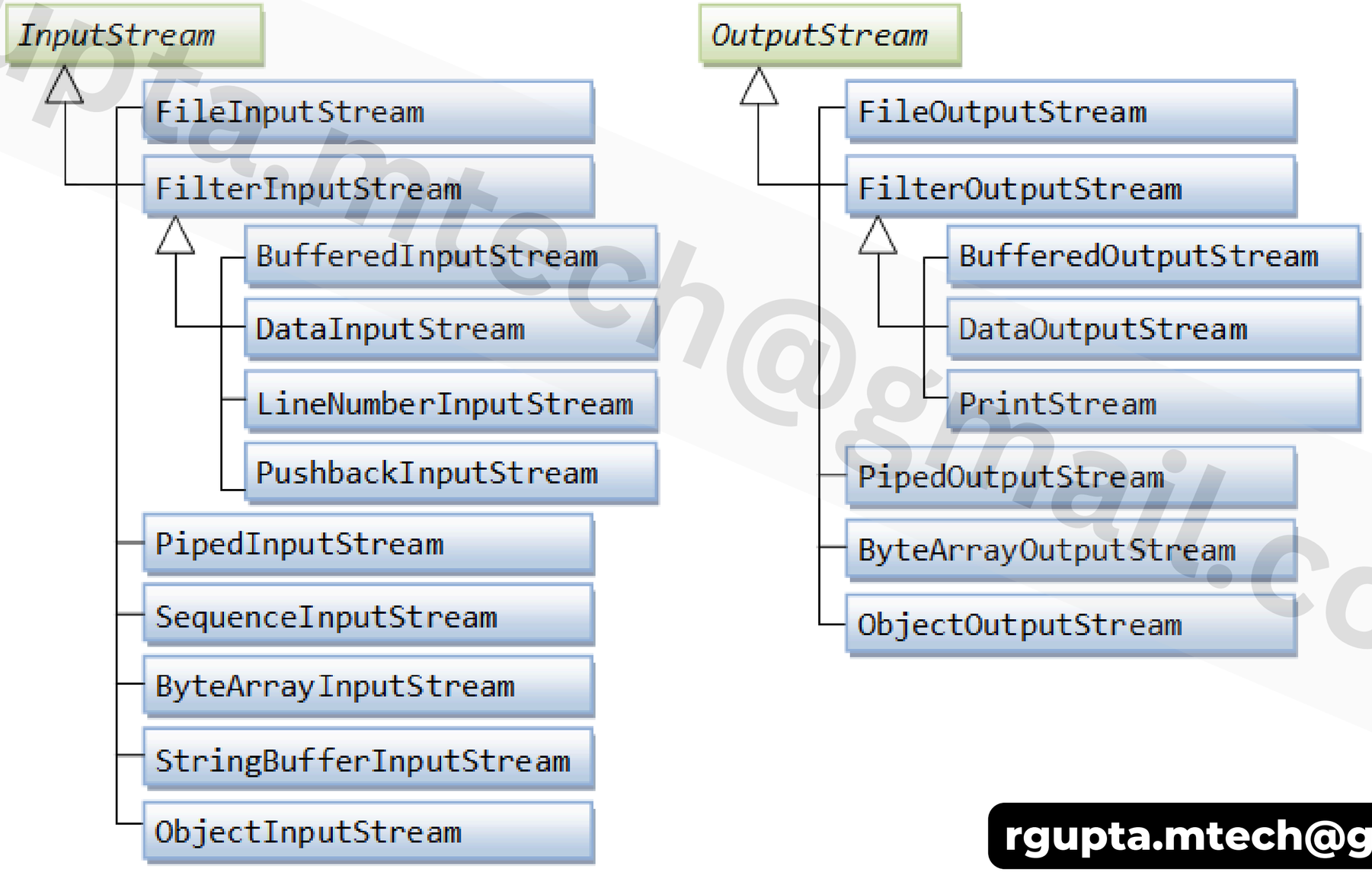
Internal Data Formats:

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

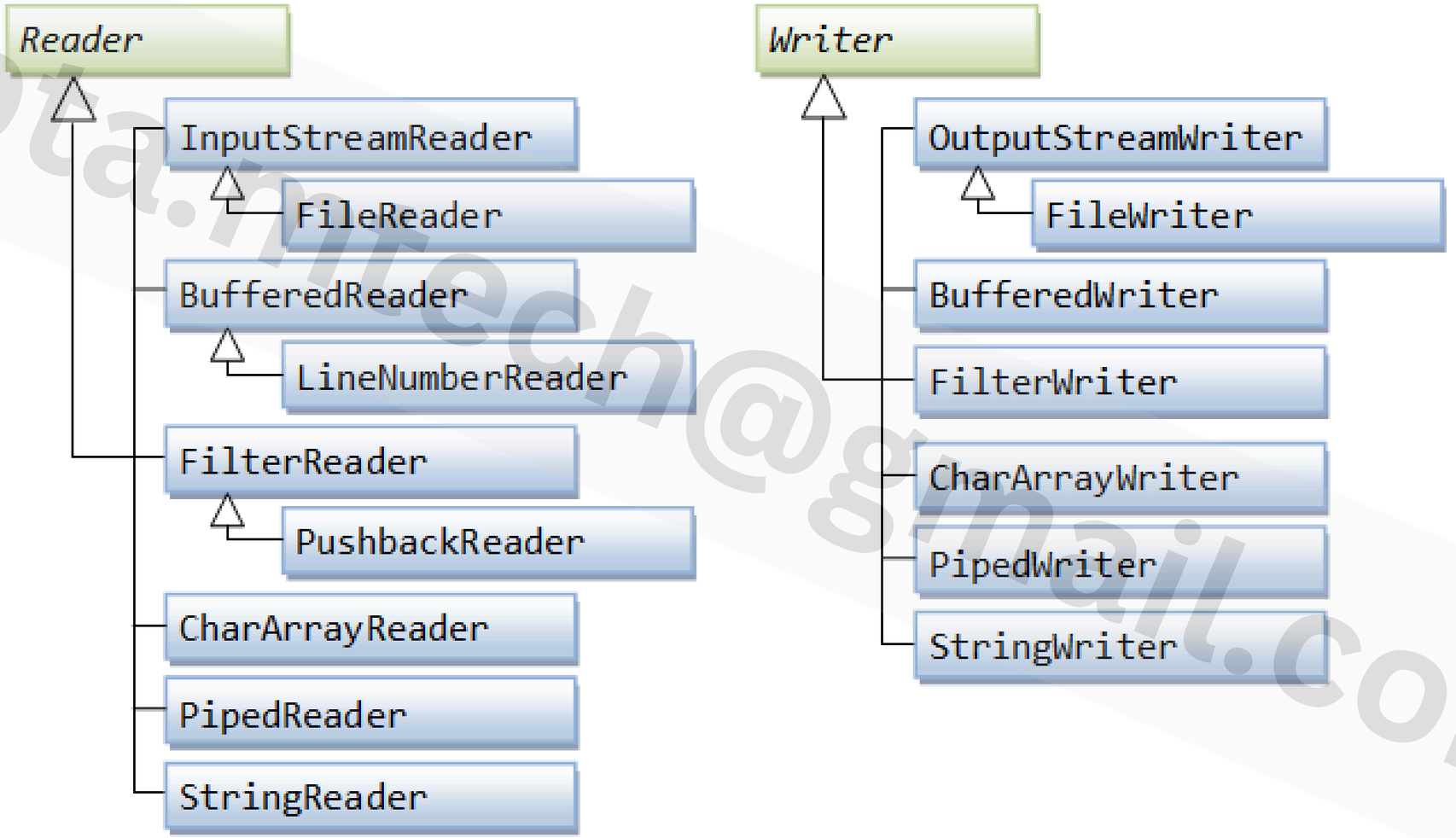
External Data Formats:

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

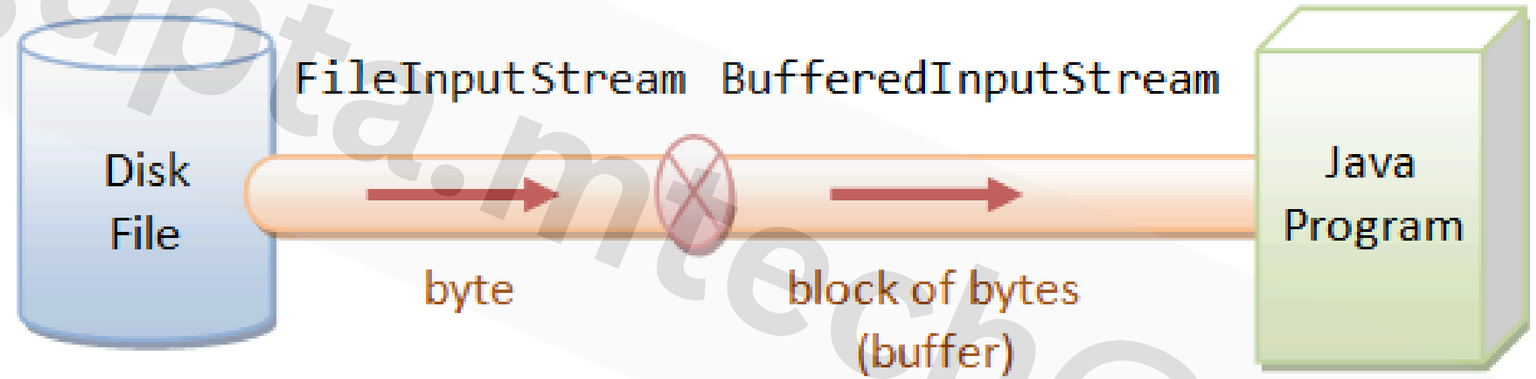
Byte-Based I/O & Byte Streams

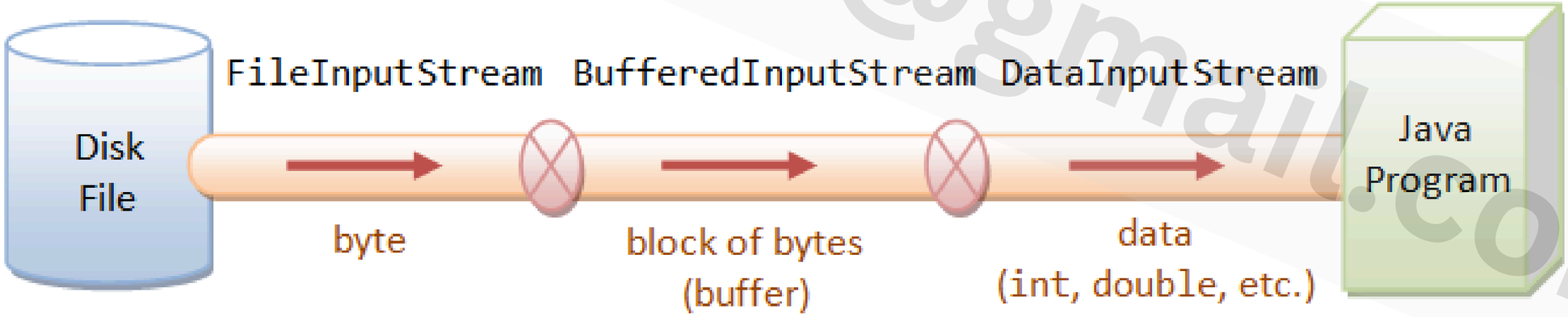


Character-Based I/O & Character Streams



Layered (or Chained) I/O Streams





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();
        }

    }

}
```

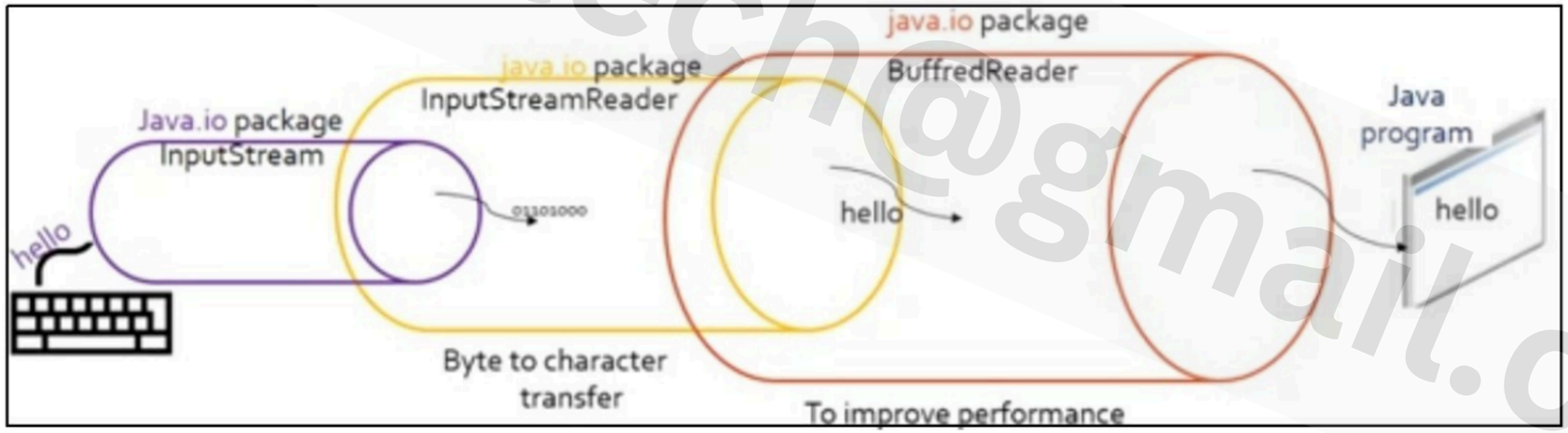
Buffered Reader and BufferedWriter

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")));
```



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");
```

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();
}
```

Day 3:

- Collections and Generics
- Multithreading
- GOF design patterns

Day-3
Session -1
Introduction to Collection API

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?

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

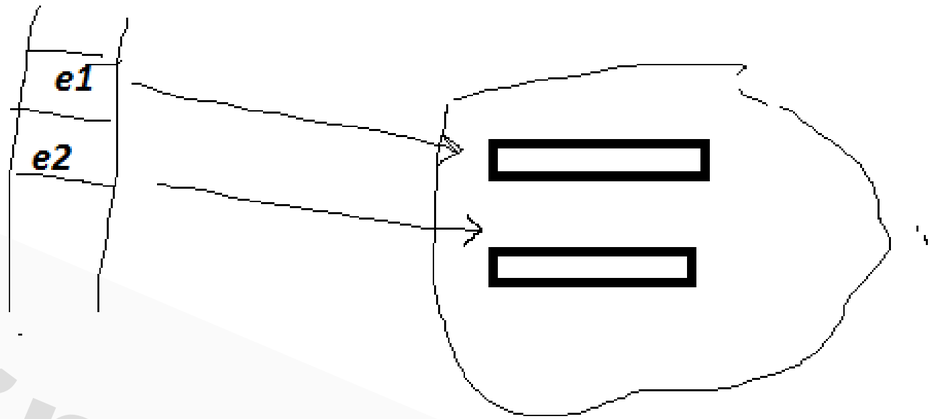
---

com.Employee@addbf1

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

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...

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;
    }
```

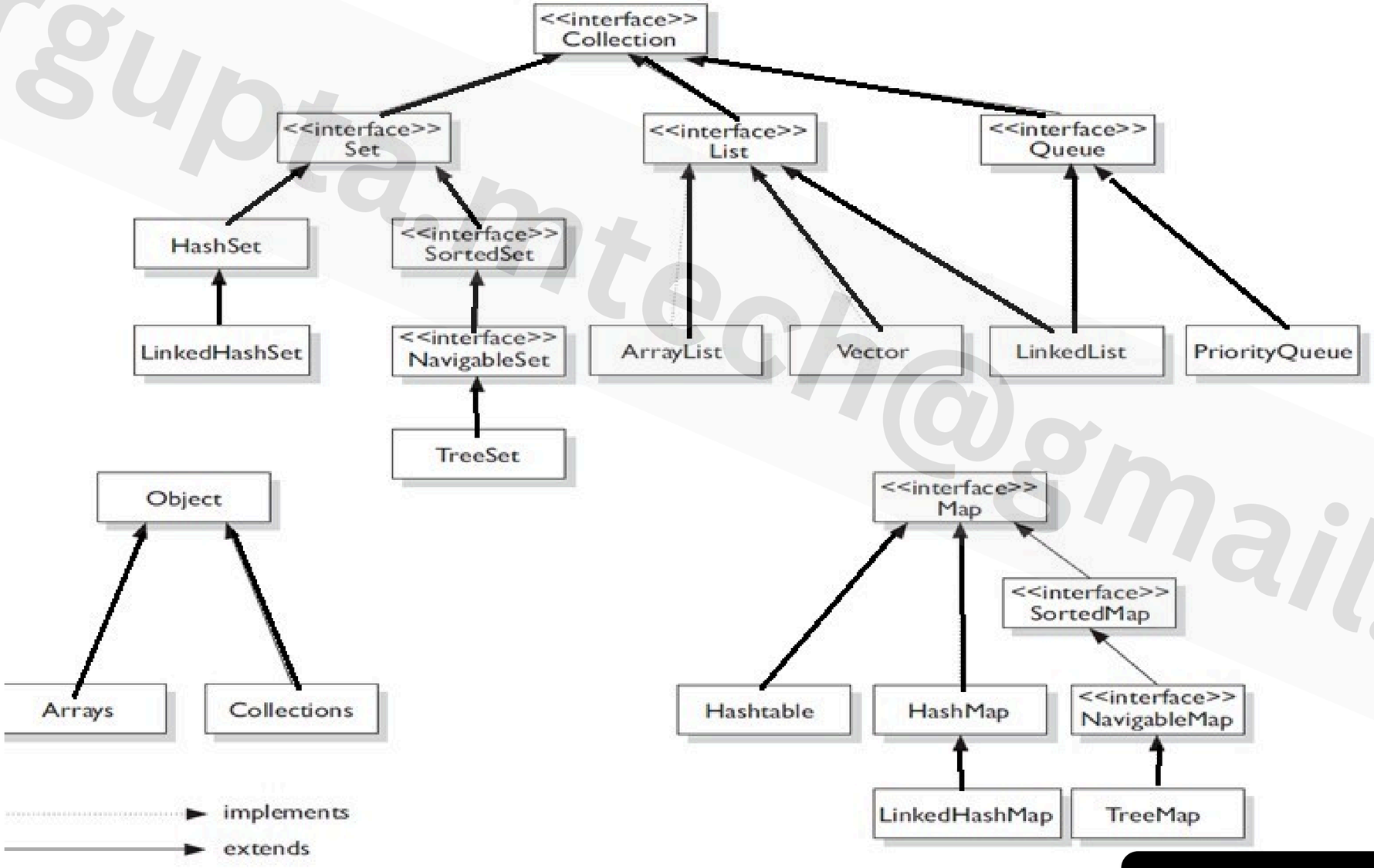
Java collection

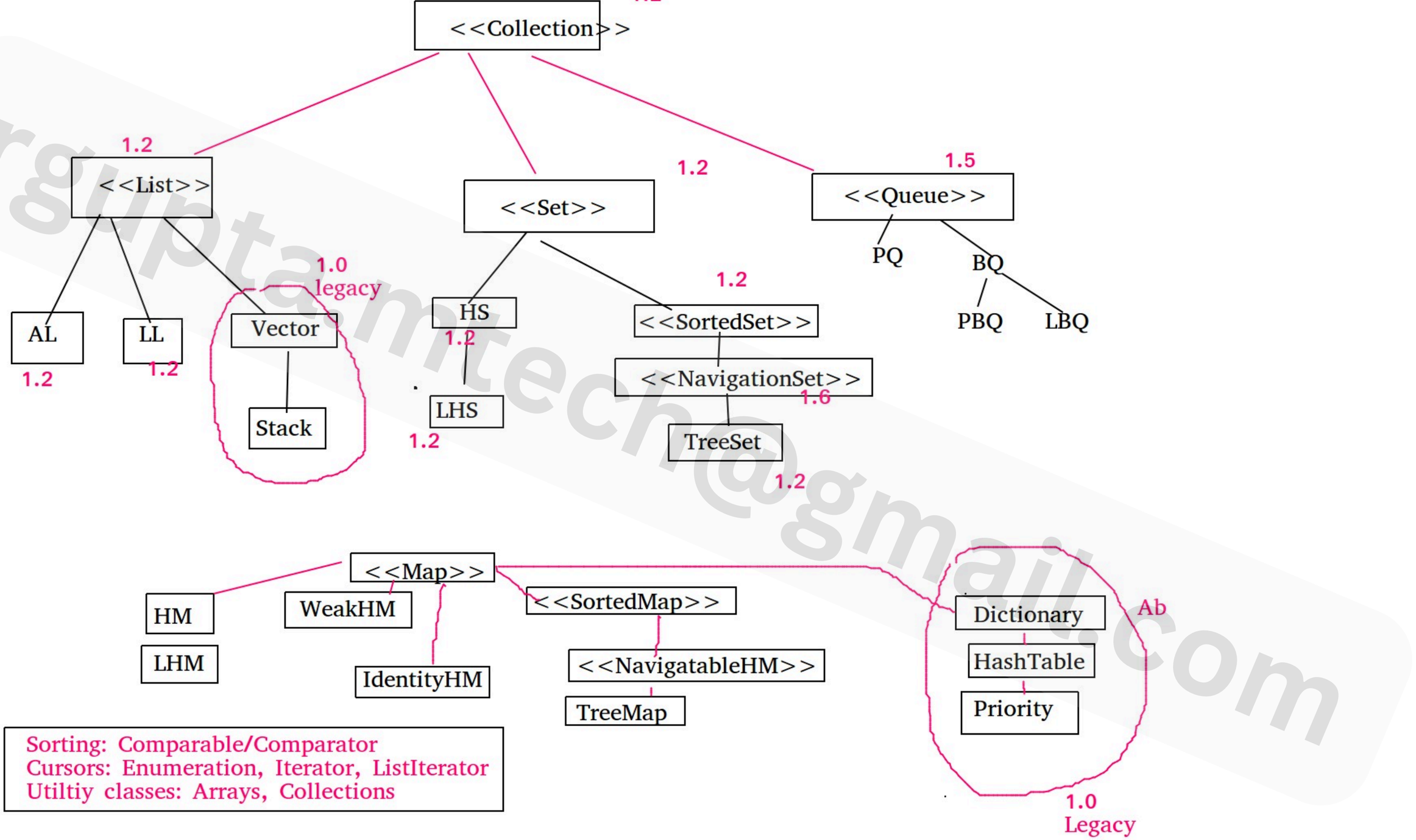
Java collections can be considered a kind of readymade data structure, we should only need to know how to use them and how they work....

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

rgupta.mtech@gmail.com

collection





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.

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");
}
```

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)

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);
    }
```

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();
}
```

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();
        }
}
```

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...**

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

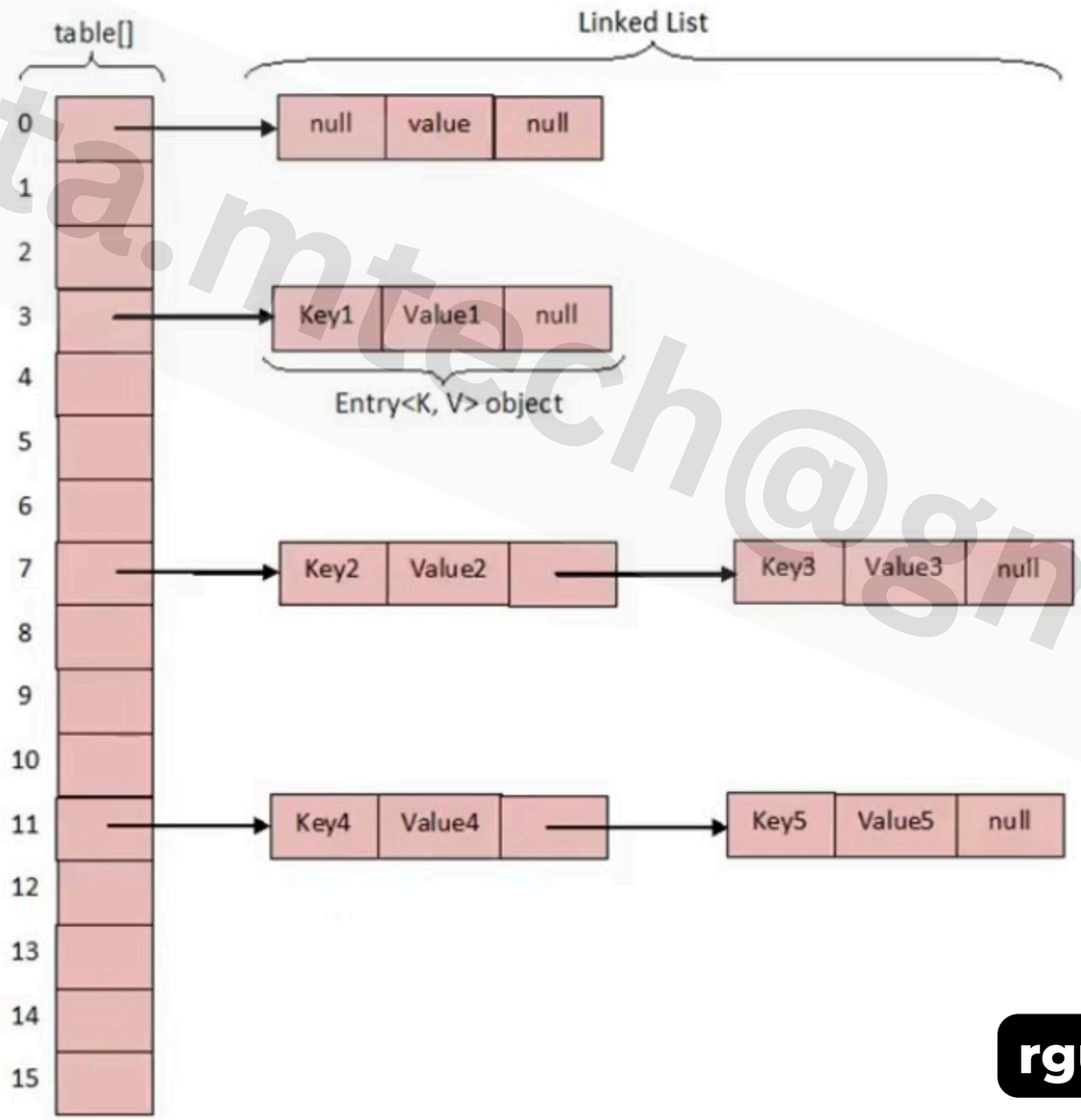
HashMap Internal Structure

- 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;

}
```

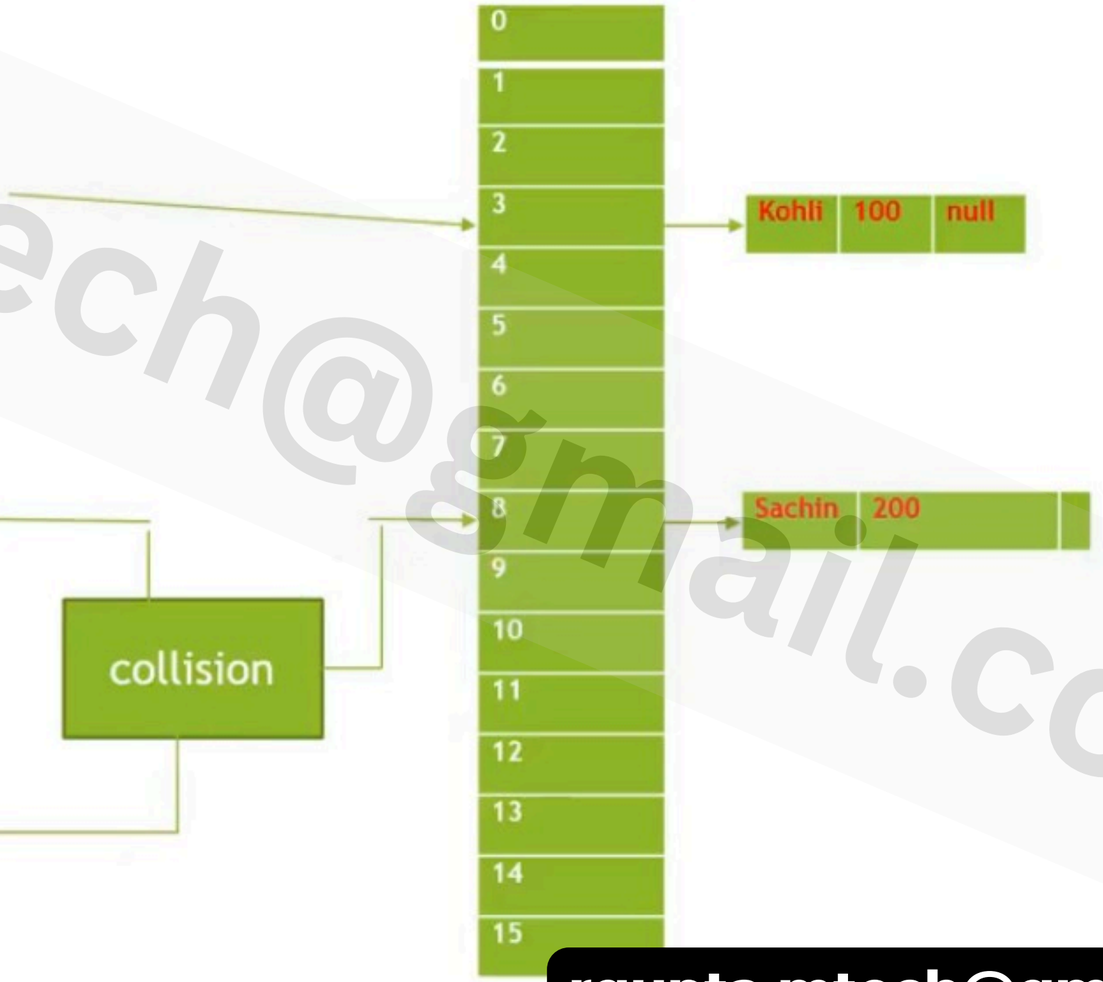
- **key** : It stores the key of an element and its final.
- **value** : It holds the value of an element.
- **next** : It holds the pointer to next key-value pair. This attribute makes the key-value pairs stored as a linked list.
- **hash** : It holds the hashcode of the key.

HashMap Internal Structure

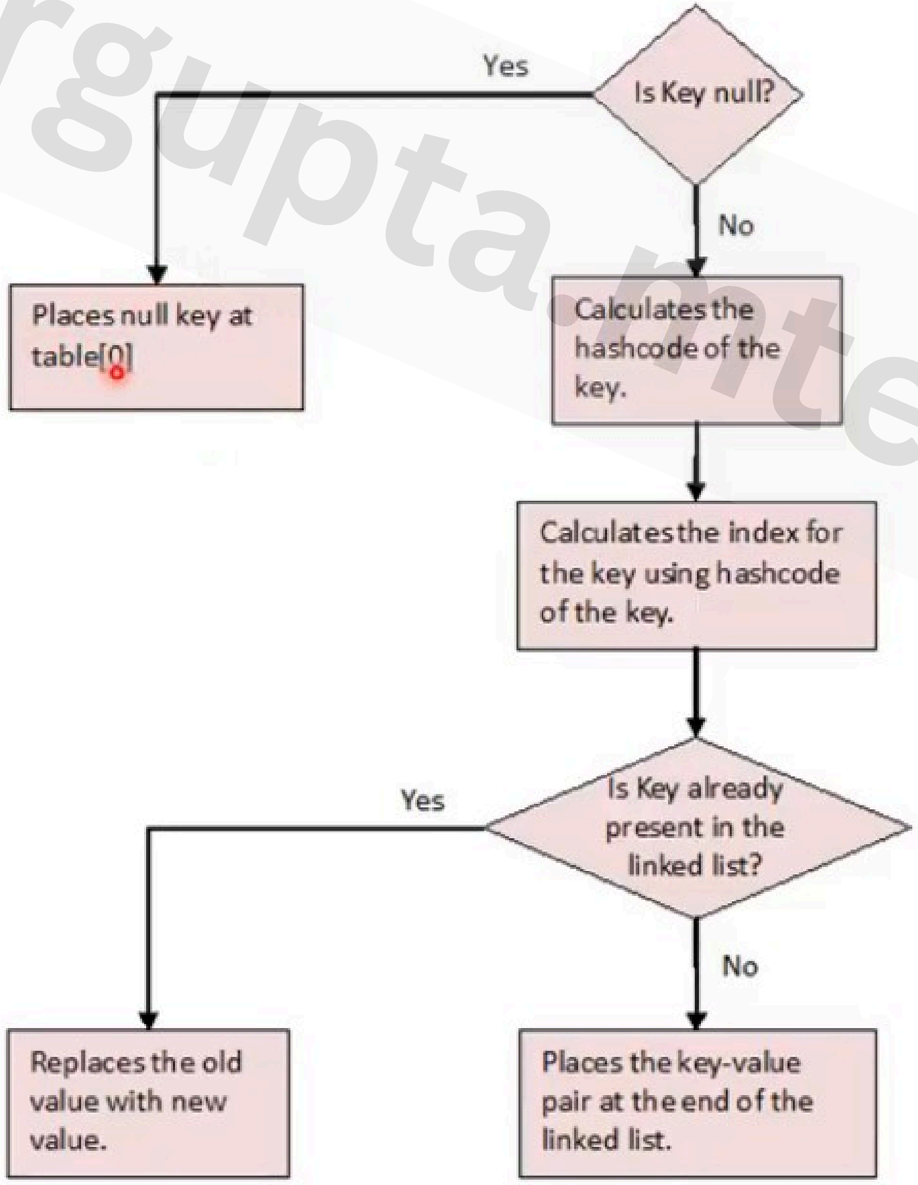


HashMap Internal Structure

```
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
```



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.

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.

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>();
```

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());

    }
}
```

Need of Generics

Before Java 1.5

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

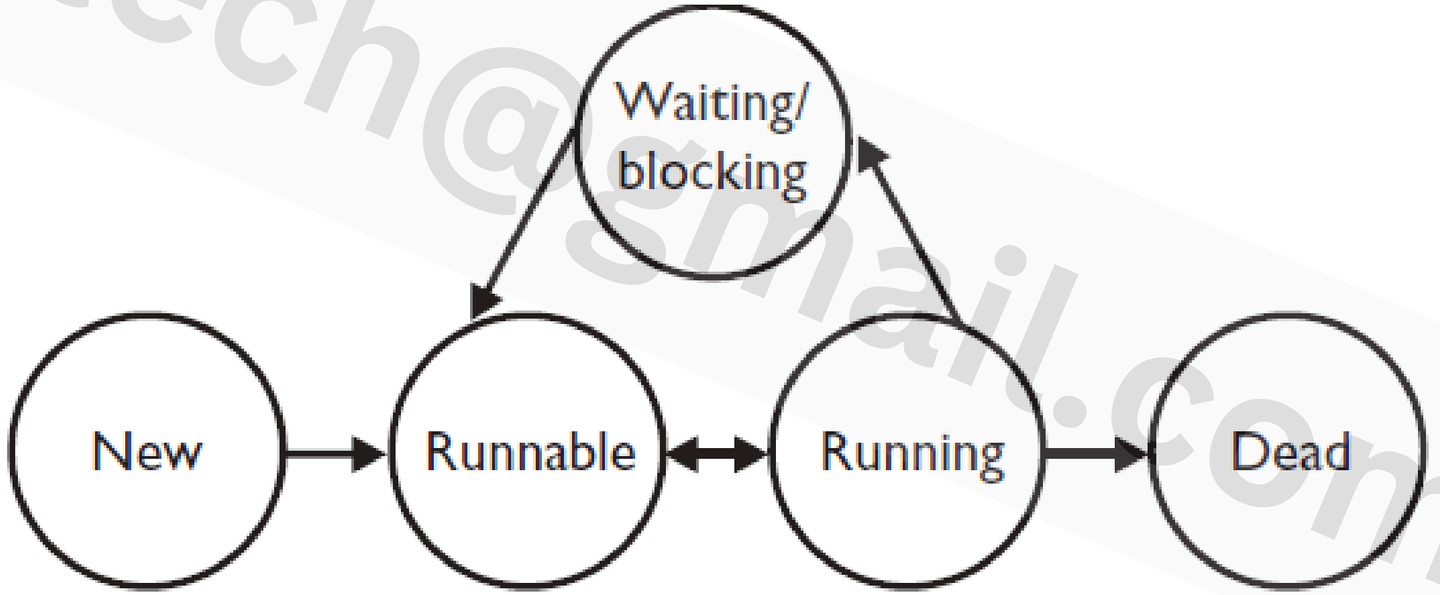
Now Java 1.5 onward
List<String>
list=new ArrayList<String>();
list.add(“foo”);//ok
list.add(22);// compile time error
Generics provide type safety
Generics is compile time phenomena…

rgupta.mtech@gmail.com

Day-3
Session -2
Introduction to Java Threads

rgupta.mtech@gmail.com

Java Threads

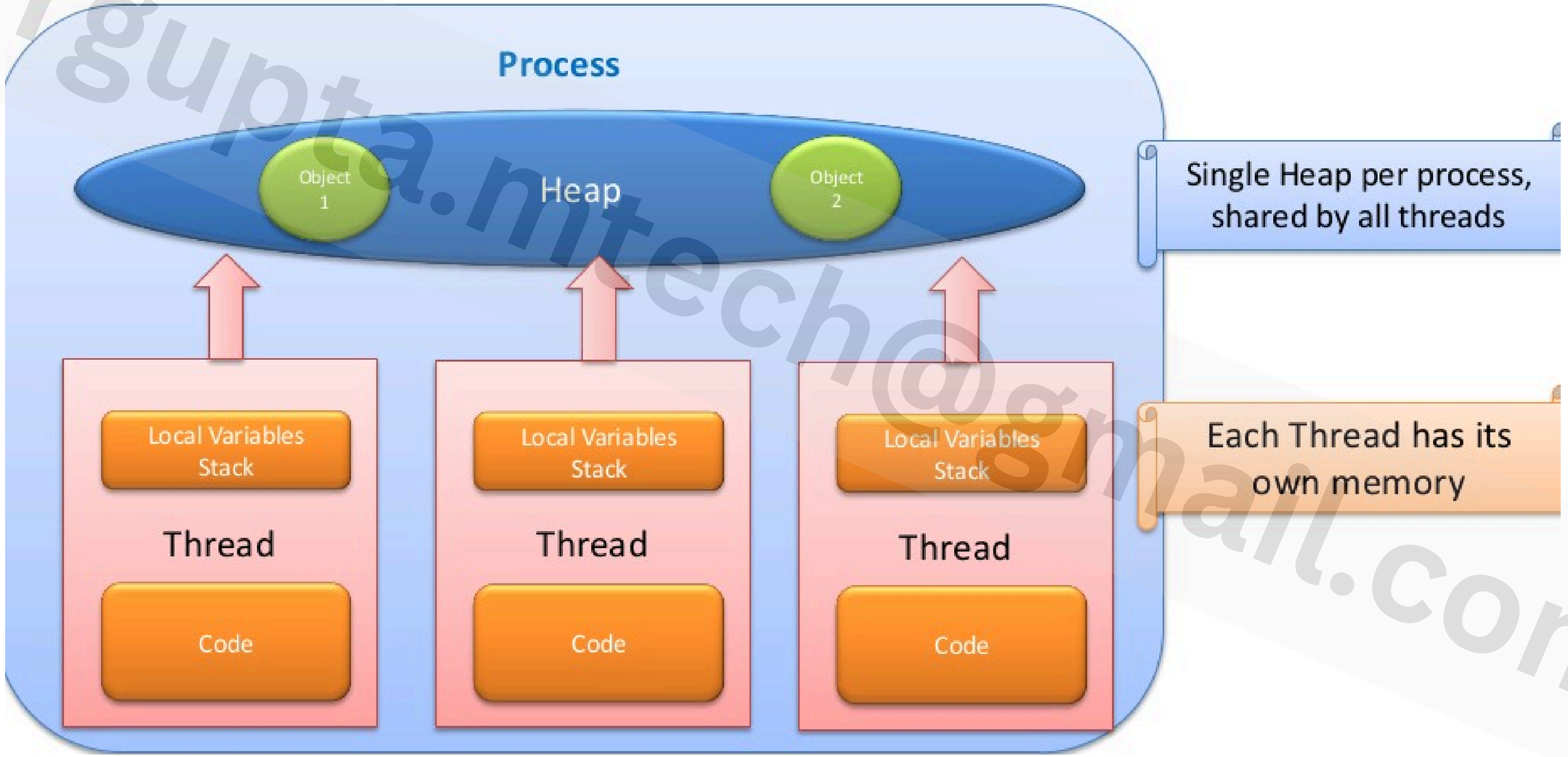


Truth about concurrency

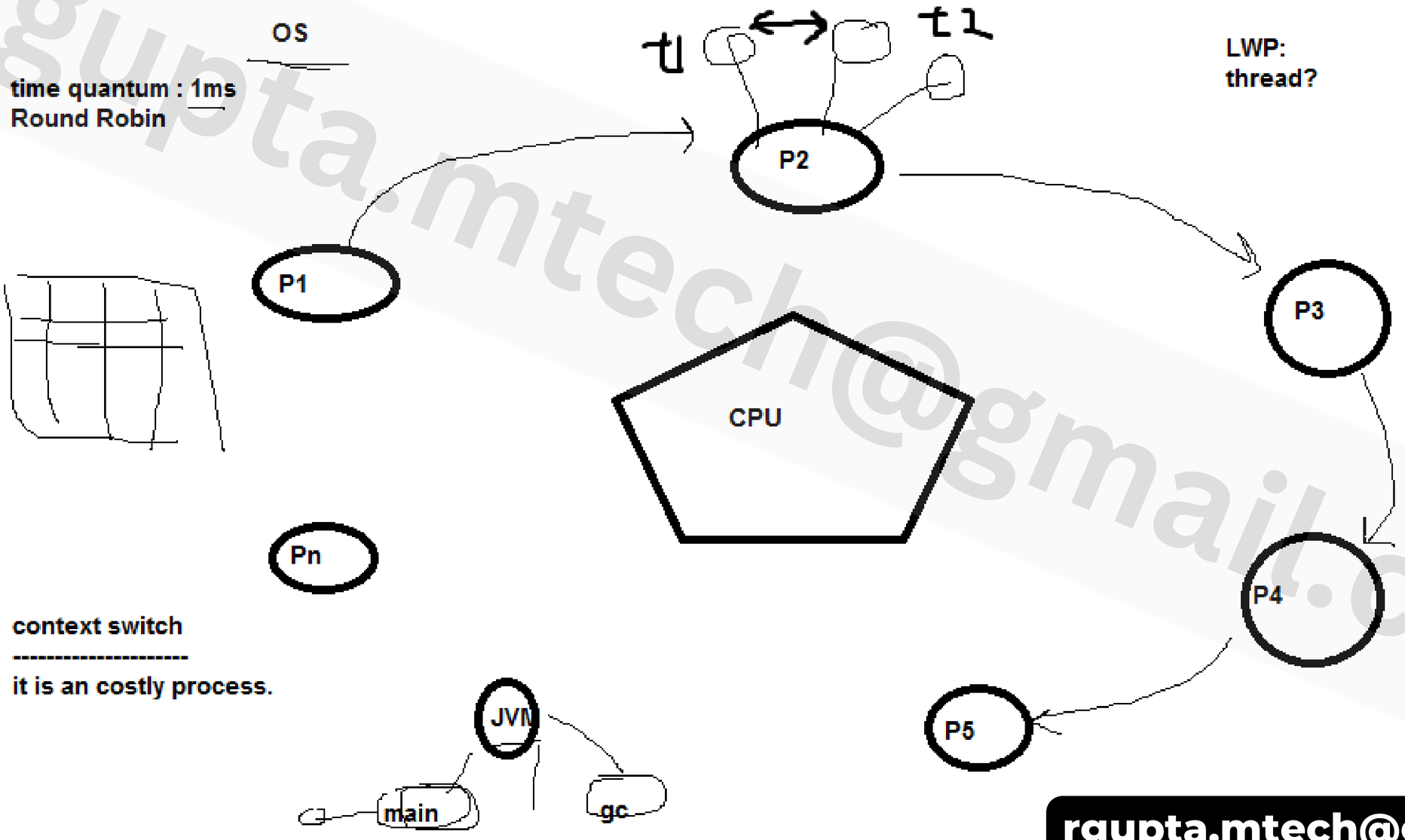
How you designed it

What happens in reality

Process and Threads



What is threads? LWP



Thread LifeCycle

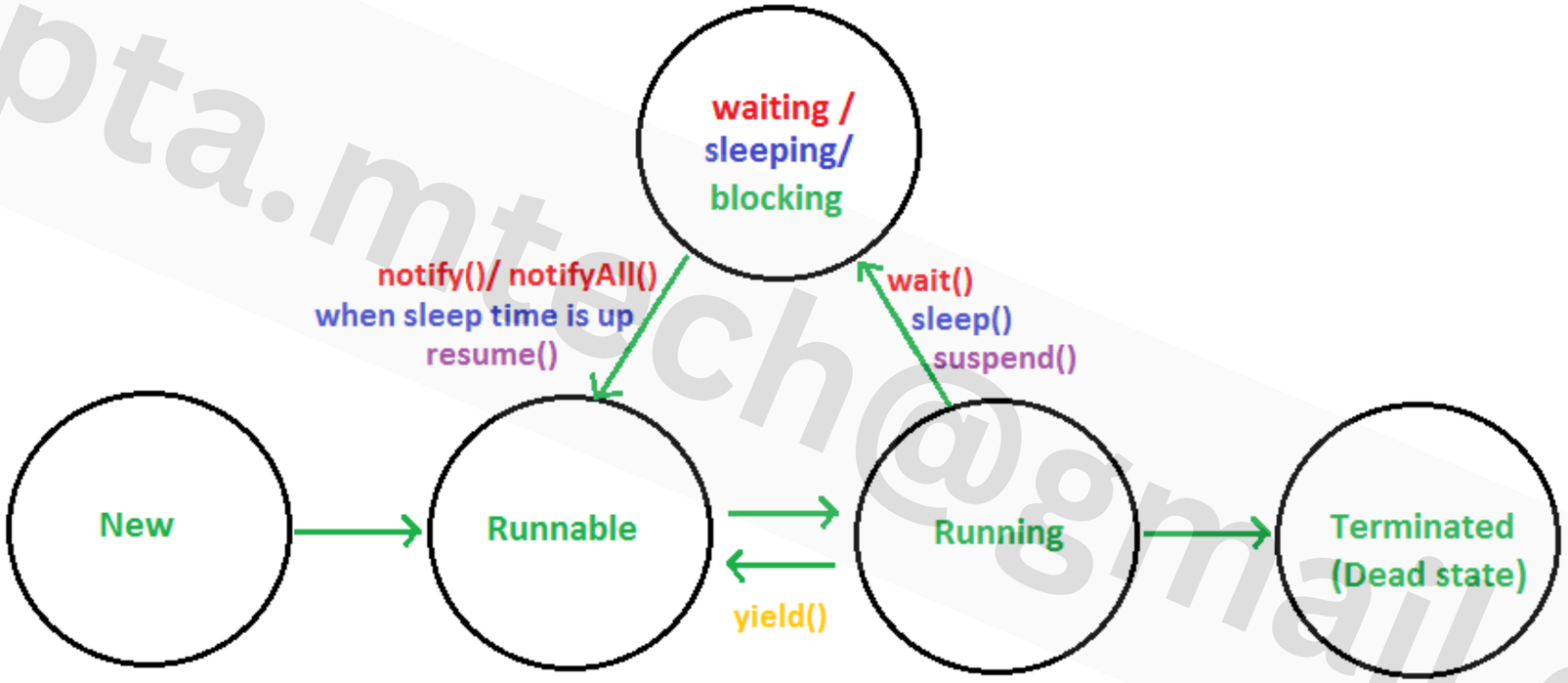


Fig. THREAD STATES

Important Methds of Thread class

- **start()**
  - Makes the Thread Ready to run
- **isAlive()**
  - A Thread is alive if it has been started and not died.
- **sleep(milliseconds)**
  - Sleep for number of milliseconds.
- **isInterrupted()**
  - Tests whether this thread has been interrupted.
- **Interrupt()**
  - Indicate to a Thread that we want to finish. If the thread is blocked in a method that responds to interrupts, an InterruptedException will be thrown in the other thread, otherwise the interrupt status is set.
- **join()**
  - Wait for this thread to die.
- **yield()**
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute.

sleep() yield() and wait()

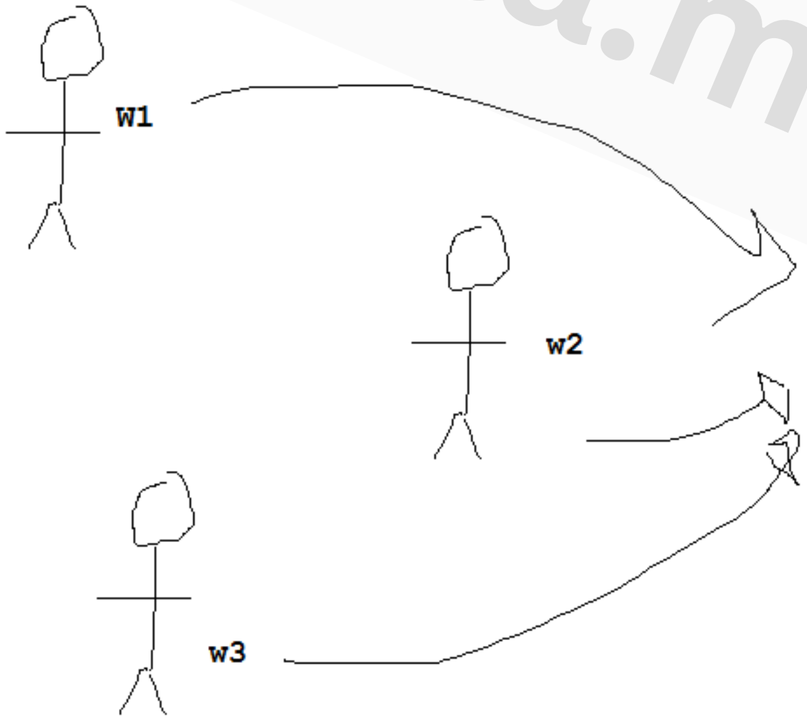
- sleep(n) says ***"I'm done with my time slice, and please don't give me another one for at least n milliseconds."*** The OS doesn't even try to schedule the sleeping thread until requested time has passed.
- yield() says ***"I'm done with my time slice, but I still have work to do."*** The OS is free to immediately give the thread another time slice, or to give some other thread or process the CPU the yielding thread just gave up.
- .wait() says ***"I'm done with my time slice. Don't give me another time slice until someone calls notify()."*** As with sleep(), the OS won't even try to schedule your task unless someone calls notify() (or one of a few other wakeup scenarios occurs).

wait() vs sleep()

Called from Synchronized block	No Such requirement
Monitor is released	Monitor is not released
Awake when notify() and notifyAll() method is called on the monitor which is being waited on.	Not awake when notify() or notifyAll() method is called, it can be interrupted.
not a static method	static method
wait() is generally used on condition	sleep() method is simply used to put your thread on sleep.
Can get *spurious wakeups* from wait (i.e. the thread which is waiting resumes for no apparent reason). We Should always wait whilst spinning on some condition , ex : *synchronized {* *while (!condition) monitor.wait();* *}*	This is deterministic.
Releases the lock on the object that wait() is called on	Thread does *not* release the locks it holds

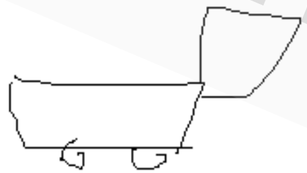
rgupta.mtech@gmail.com

Creating threads in Java?



Thread
---
consider object of threads as worker

Implementation of Runnable as Job



simulation

Sleep()

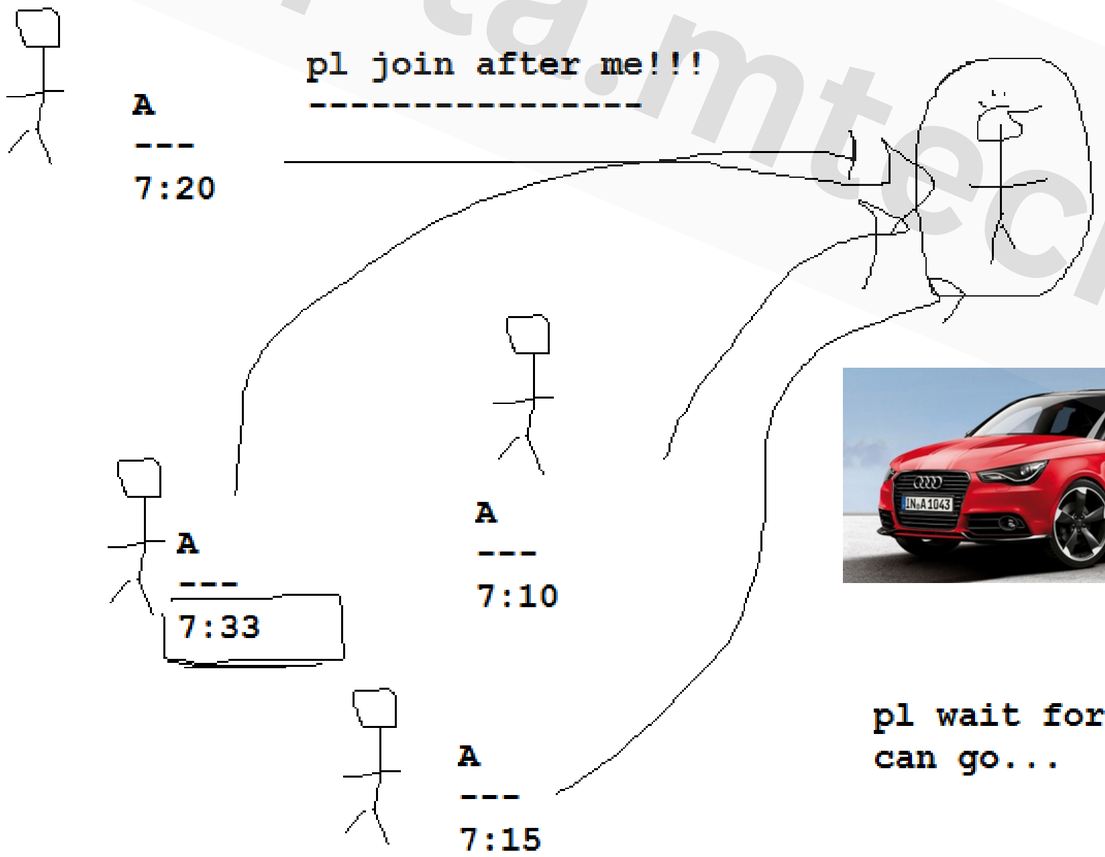
Job

```java
class Job implements Runnable{


    public void run() {
        // TODO Auto-generated method stub
    }

}
```

```java
class MyThread extends Thread{
    public void run() {
        // TODO Auto-generated method stub
    }
}
```


rgupta.mtech@gmail.com

Understanding join() method



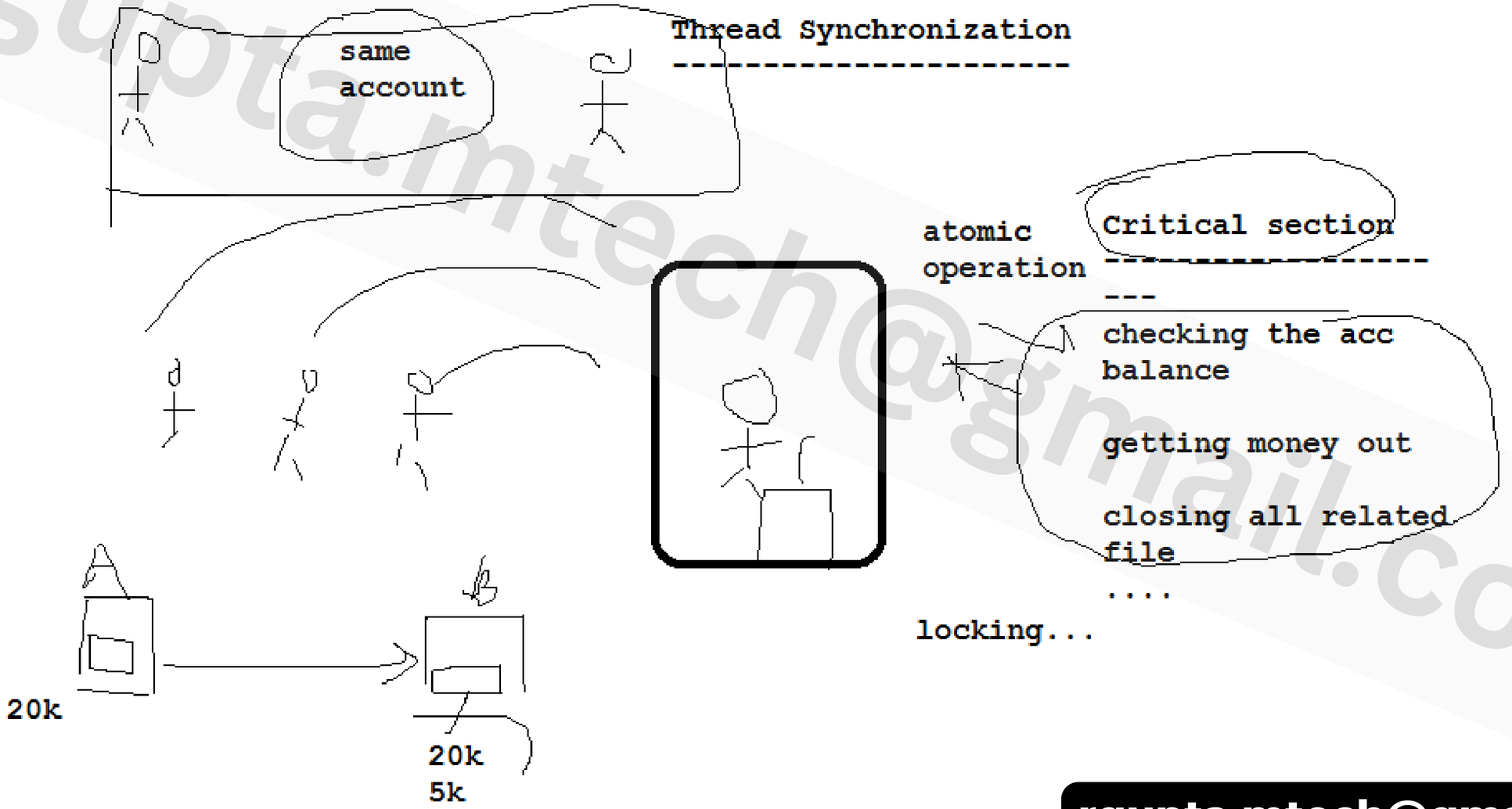
pl wait for me for 5 min else u
can go...

Checking thread priorities

```
class Clicker implements Runnable
{
     int click=0;
     Thread t;
     private volatile boolean running=true;
     public  Clicker(int p)
     {
          t=new Thread(this);
          t.setPriority(p);
     }
     public void run()
     {
          while(running)
               click++;
     }
     public void stop()
     {
          running=false;
     }
     public void start()
     {
          t.start();
     }
}
```

```
.....
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
Clicker hi=new Clicker(Thread.NORM_PRIORITY+2);
Clicker lo=new Clicker(Thread.NORM_PRIORITY-2);
lo.start();
hi.start();
try
{
     Thread.sleep(10000);
}
catch(InterruptedException ex){}
lo.stop();
hi.stop();
//wait for child to terminate
try
{
     hi.t.join();
     lo.t.join();
}
catch(InterruptedException ex)
{

}
System.out.println("Low priority thread:"+lo.click);
System.out.println("High priority thread:"+hi.click);
....
....
```

Understanding thread synchronization



Synchronization

Synchronization

- **Mechanism to controls the order in which threads execute**
- **Competition vs. cooperative synchronization**

Mutual exclusion of threads

- **Each synchronized method or statement is guarded**
- **by an object.**
- **When entering a synchronized method or statement,**
- **the object will be locked until the method is finished.**
- **When the object is locked by another thread, the**
- **current thread must wait.**

Synchronized Keyword

- Synchronizing instance method

```
class SpeechSynthesizer {
    synchronized void say( String words ) {
        // speak
    }
}
```

- Synchronizing multiple methods.

```
class SpreadSheet {
    int cellA1, cellA2, cellA3;

    synchronized int sumRow() {
        return cellA1 + cellA2 + cellA3;
    }

    synchronized void setRow( int a1, int a2, int a3 ) {
        cellA1 = a1;
        cellA2 = a2;
        cellA3 = a3;
    }
    ...
}
```

- Synchronizing a block of code.

```
synchronized ( myObject ) {
    // Functionality that needs exclusive access to resources
}
```

```
synchronized void myMethod () {
    ...
}
```

is equivalent to:

```
void myMethod () {
    synchronized ( this ) {
        ...
    }
}
```

Using thread synchronization

```
class CallMe
{
      synchronized void call(String msg)
      {
            System.out.print("["+msg);
            try
            {
                  Thread.sleep(500);
            }
            catch(InterruptedException ex){}
            System.out.println("]");
      }
}
```

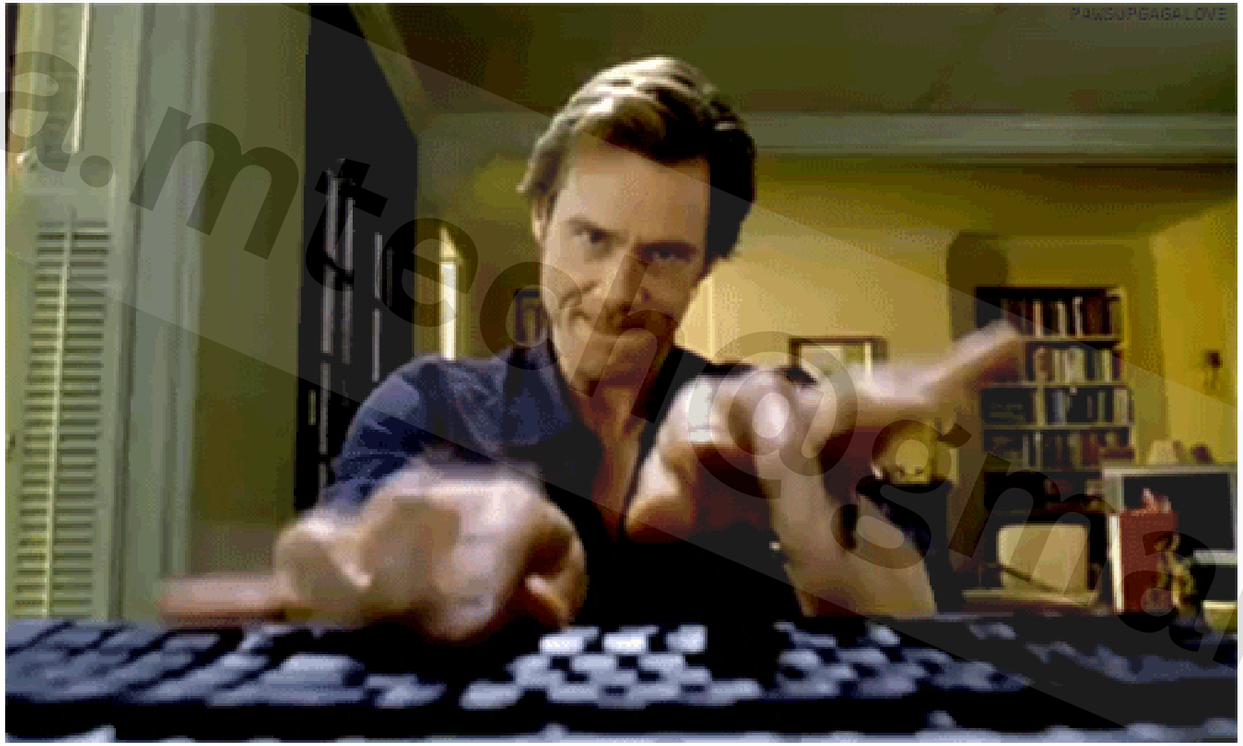
```
class Caller implements Runnable
{
      String msg;
      CallMe target;
      Thread t;
      public Caller(CallMe targ,String s)
      {
            target=targ;
            msg=s;
            t=new Thread(this);
            t.start();
      }
      public void run()
      {
            target.call(msg);
      }
}
```

```
Caller ob1=new Caller(target, "Hello");
Caller ob2=new Caller(target,"Synchronized");
Caller ob3=new Caller(target,"Java");
```

Day 3:
Session-3
Introduction to SOLID & GOF

rgupta.mtech@gmail.com

Programmer are Not typist but thinker



Learning OOPs, SOLID, Design Patterns step by step

Lib & Framework

BTech/MTech

12th class

Design patterns

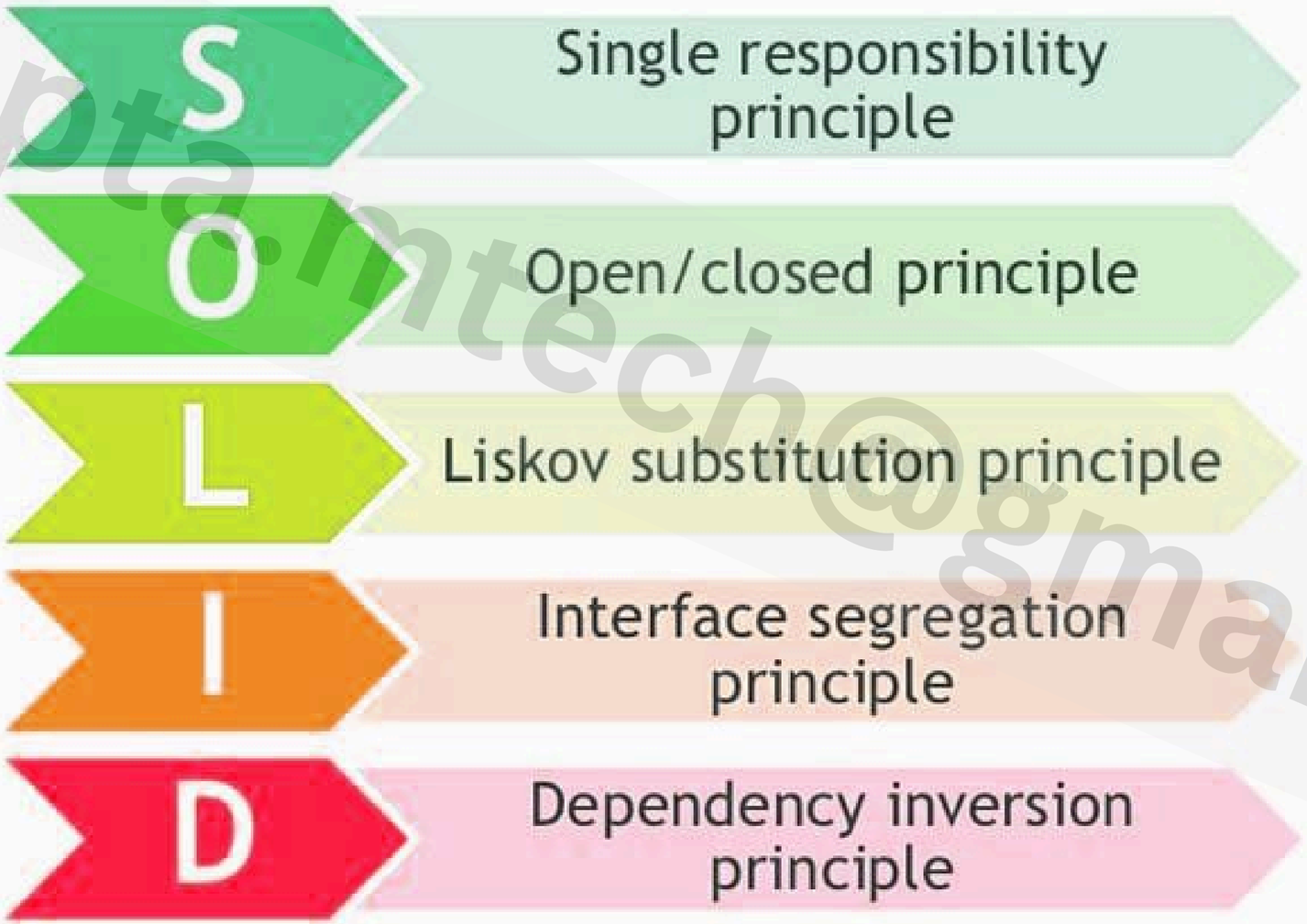
10th class

SOLID

5th class

clear understanding of OOPs concepts

SOLID Principles



Without Good design our application would be...

Rigid

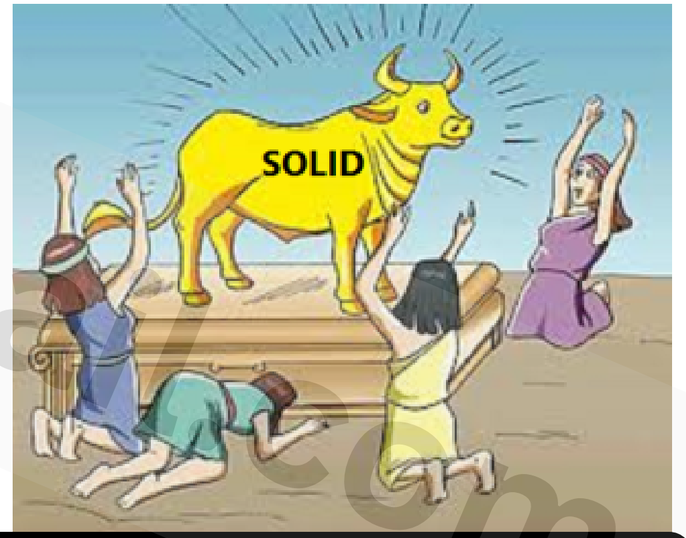
Fragile

Immobile

Making everyone's life miserable

SOLID Principles

- In software engineering **SOLID** is an acronym for **5** design principles
- These principles are mainly promoted by **Robert C. Martin** in his book back in **2000**
- **It helps us to write code that is ...**
  - **Loosely coupled**
  - **Highly cohesive**
  - **Easily composable**
  - **Reusable**



Busy Coder Academy

S - Single Responsibility Principle

- Every single software entity (**class** or **method**) should have only a **single reason** to change
- If a given class (or method) does **multiple operations** then it is advisable to separate into distinct classes (or methods)
- If there are **2** reasons to change a given class then it is a sign of violating the single responsibility principle

”There should be **NEVER** be more than **ONE** reason for a class to change”

Busy Coder Academy

O – Open/Closed Principle

- Software entities should be **open for extension** and closed for modification
- We have to design every new module such that if we add a new behavior then we **do not have to change the existing modules**
- CLOSELY RELATED TO SINGLE RESONSIBILITY PRINCIPLE
- a class should not extend an other class explicitly – we should define a **common interface** instead
- we can change the classes at **runtime** due to the common interface

”Modules must be **OPEN** for extension, **CLOSED** for modification”

L – Liskov Substitution Principle

We extend some classes and creating some **derived classes**

It would be great if the new derived classed would work as well **without replacing the functionality** of the classes otherwise the new classes can produce undesired effects when they are used in **existing program modules**

*"Objects of a superclass shall be replaceable with objects of its subclasses **without breaking** the application"*

I – Interface Segregation Principle

- What is the motivation behind **interface segregation principle**?

WE USE SEVERAL INTERFACES OR ABSTRACT CLASSES
IN ORDER TO ACHIEVE ABSTRACTION

- Sometimes we want to implement that interface but just for the sake of some methods defined in by that interface we can end up with **fat interfaces** – containing more methods than the actual class needs

"Clients should not depend upon the interfaces they do not use"

How can we pollute the interfaces?
OR
How do we end up creating Fat interfaces?

D – Dependency Inversion Principle

- What is the motivation behind **dependency inversion principle**?
- USUALLY THE LOW LEVELS MODULES RELY HEAVILY ON HIGH LEVEL MODULES (BOTTOM UP SOFTWARE DEVELOPMENT)
  - When implementing an application usually we start with the **low level** software components then we implement the **high level** modules that rely on these low level modules

"High level modules should not depend on the low level details modules, instead both should depend on abstractions"

Design Patterns

design patterns *are more about how to design your code*

+ concrete implementations of the design principles

design principles *(SOLID principles) allow scalable and maintainable software architectures*

Top 10 Object Oriented Design Principles

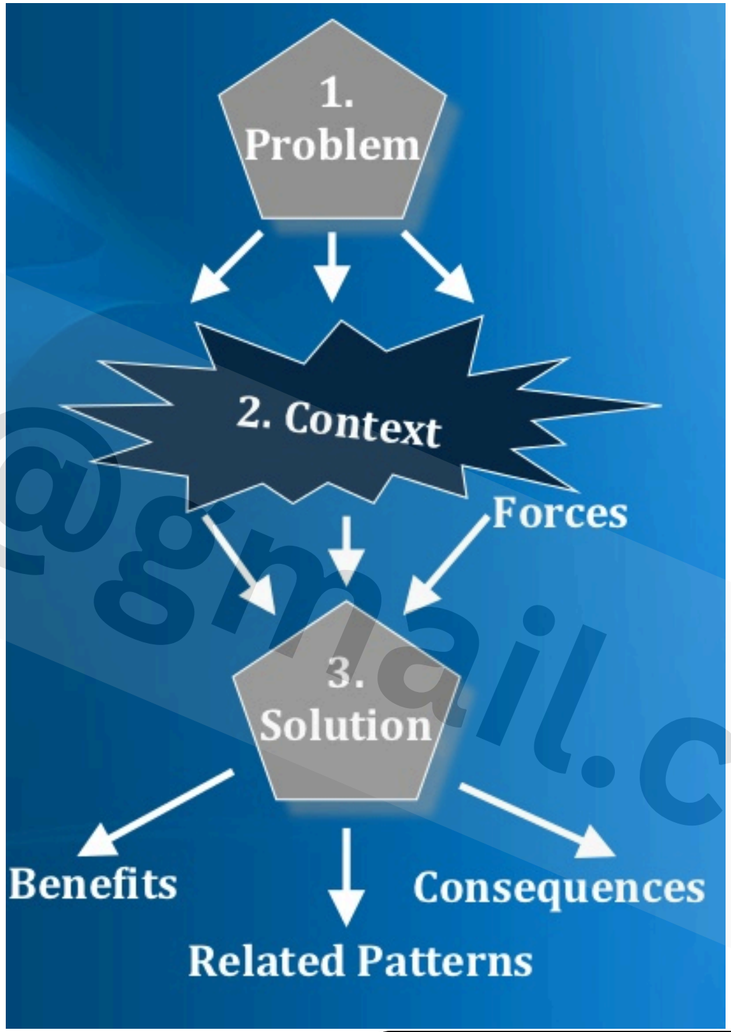
1. DRY (Don't repeat yourself) - avoids duplication in code.
2. Encapsulate what changes - hides implementation detail, helps in maintenance
3. Open Closed design principle - open for extension, closed for modification
4. SRP (Single Responsibility Principle) - one class should do one thing and do it well
5. DIP (Dependency Inversion Principle) - don't ask, let framework give to you
6. Favor Composition over Inheritance - Code reuse without cost of inflexibility
7. LSP (Liskov Substitution Principle) - Sub type must be substitutable for super type
8. ISP (Interface Segregation Principle) - Avoid monolithic interface, reduce pain on client side
9. Programming for Interface - Helps in maintenance, improves flexibility
10. Delegation principle - Don't do all things by yourself, delegate it

Design Patterns

Proven way of doing Things, Gang of 4 design patterns

- Total 23 patterns

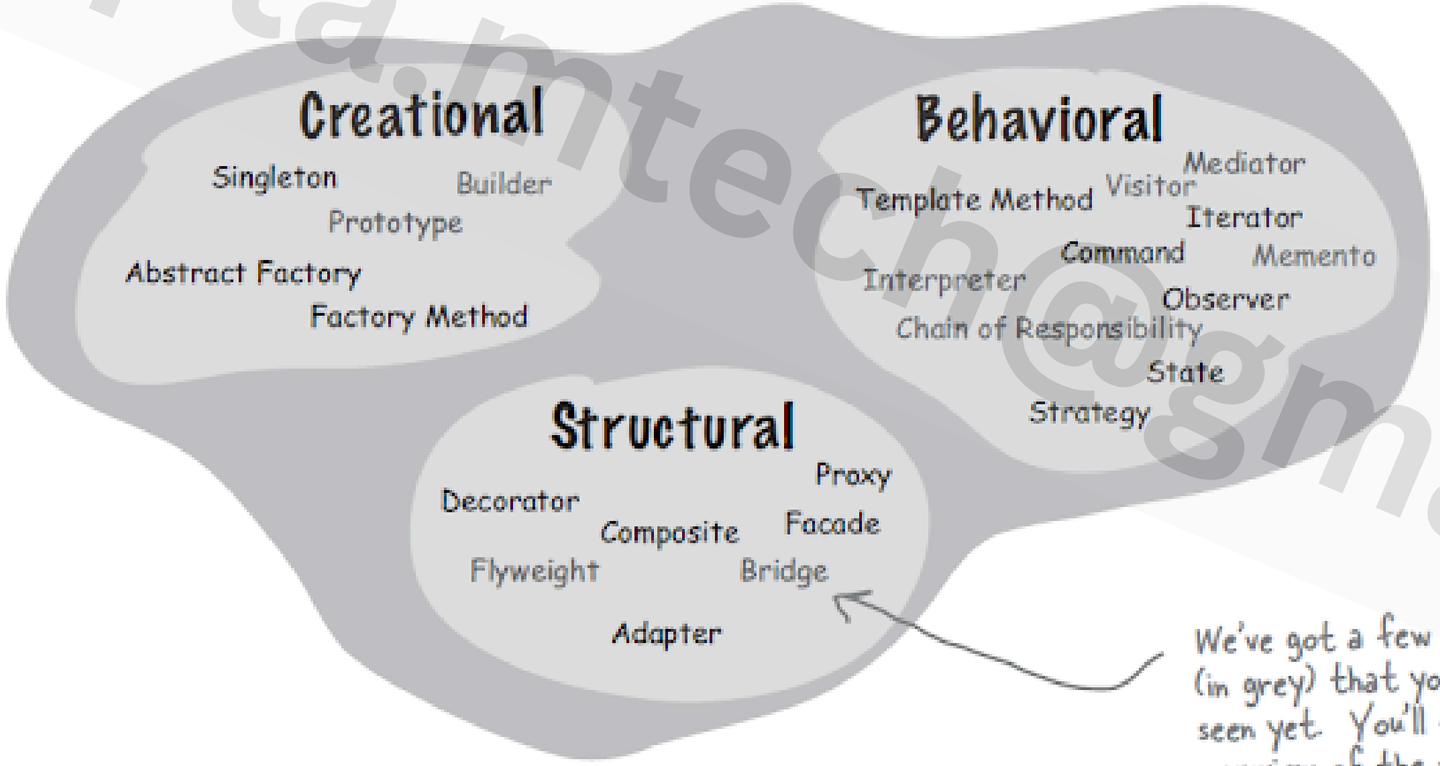
- Classification patterns
- Creational
- Structural
- Behavioral



Design Patterns

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

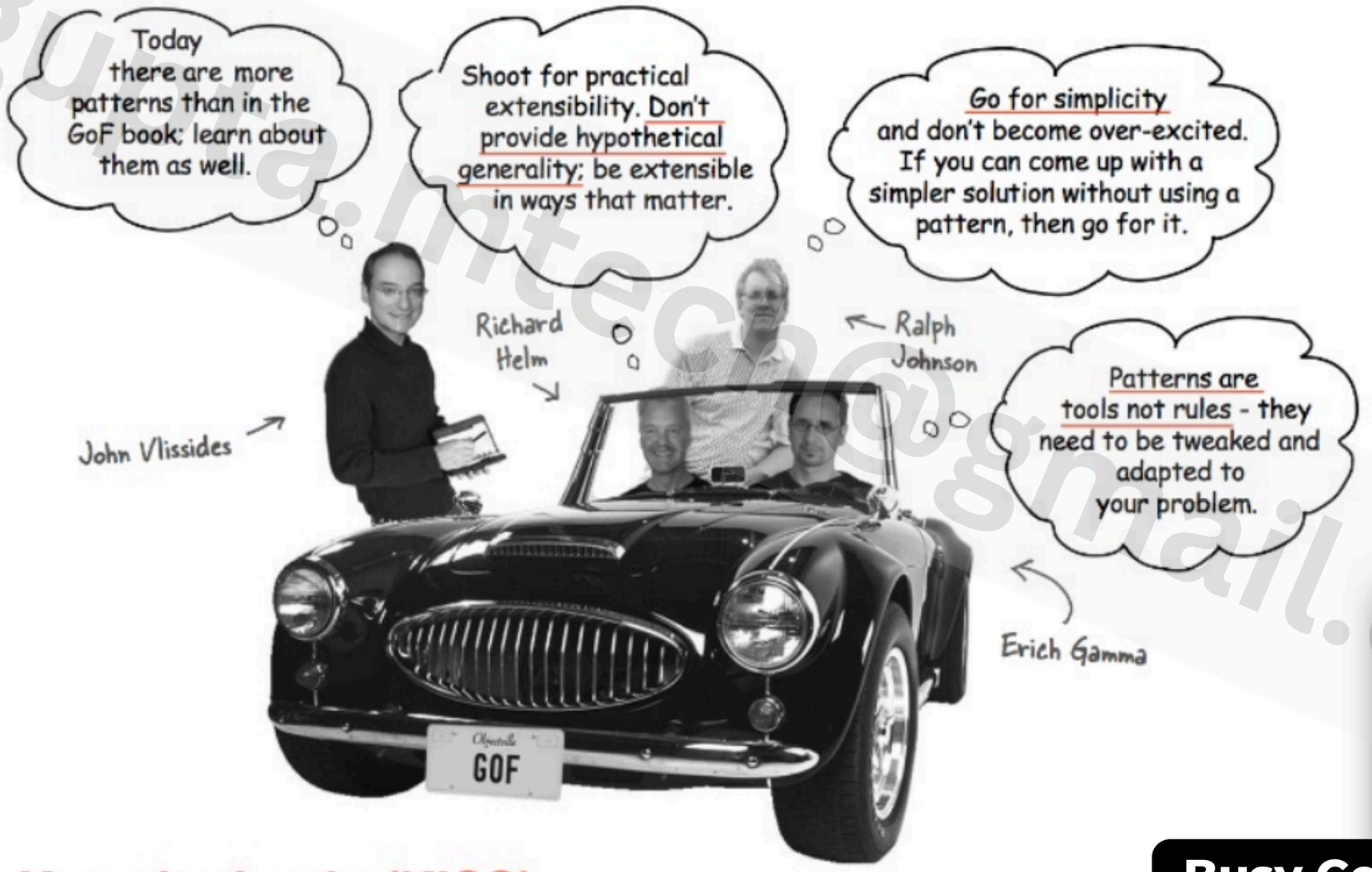
Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.



We've got a few patterns (in grey) that you haven't seen yet. You'll find an overview of the these patterns in the appendix.

Structural patterns let you compose classes or objects into larger structures.

Design Patterns



Keep it simple (KISS)

Busy Coder Academy

Design Patterns- Classification

Structural Patterns

- 1. Decorator
- 2. Proxy
- 3. Bridge
- 4. Composite
- 5. Flyweight
- 6. Adapter
- 7. Facade

Creational Patterns

- 1. Prototype
- 2. Factory Method
- 3. Singleton
- 4. Abstract Factory
- 5. Builder

Behavioral Patterns

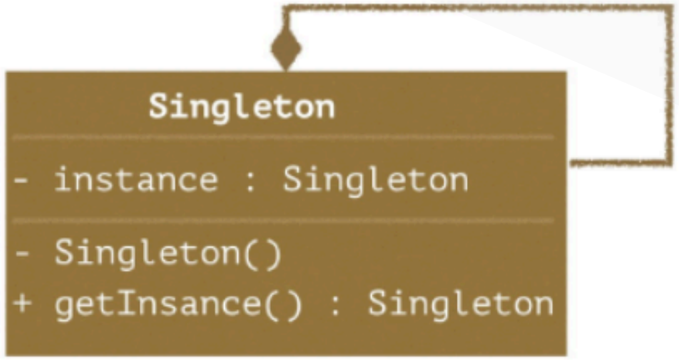
- 1. Strategy
- 2. State
- 3. TemplateMethod
- 4. Chain of Responsibility
- 5. Command
- 6. Iterator
- 7. Mediator
- 8. Observer
- 9. Visitor
- 10. Interpreter
- 11. Memento

Busy Coder Academy

Singleton Pattern

- **Singleton** pattern is a creational design pattern
- It lets you ensure that a **class has only one instance** while providing a global access point to this instance
- It ensures that a given class has just a single instance
- The singleton pattern provides a **global access point** to that given instance

"Ensure that a class has only one instance and provide a global point of access to it."



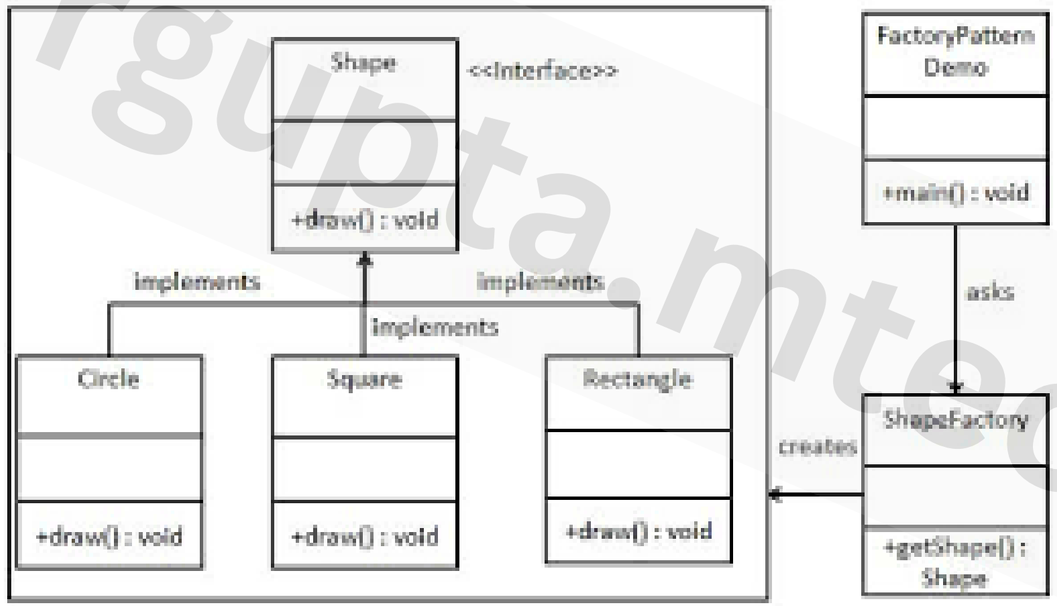
- `java.lang.Runtime#getRuntime()`
- `java.awt.Desktop#getDesktop()`
- `java.lang.System#getSecurityManager()`

❖ **Java Runtime** class is used *to interact with java runtime environment*.
❖ Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc.
❖ There is only one instance of java.lang.Runtime class is available for one java application.
❖ The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

Singleton Design Consideration

- Eager initialization
- Static block initialization
- Lazy Initialization
- Thread Safe Singleton
- Serialization issue
- Cloning issue
- Using Reflection to destroy Singleton Pattern
- Enum Singleton
- Best programming practices

Factory design pattern



Factory(Simplified version of Factory Method) - **Creates objects without exposing the instantiation logic to the client** and **Refers to the newly created object through a common interface.**

- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`
- `java.text.NumberFormat#getInstance()`

Factory Method - **Defines an interface for creating objects, but let subclasses to decide which class to instantiate** and **Refers to the newly created object through a common interface.**

Busy Coder Academy

Builder Pattern

telescoping constructors

```
class Person {

private int age;
private Gender gender;
private String dateOfBirth;
private String firstName;
private String lastName;
private String nameOfMother;
private Address address;
private PhoneNumber phoneNumber;

public Person(int age, Gender gender, String dateO
        String lastName, String nameOfMother,
        Address address, PhoneNumber phoneNumber)
    super();
    this.age = age;
    this.gender = gender;
    this.dateOfBirth = dateOfBirth;
    this.firstName = firstName;
    this.lastName = lastName;
    this.nameOfMother = nameOfMother;
```

what is the **motivation** behind builder pattern?

The Builder pattern can be used to ease the construction of a complex object from simple objects.

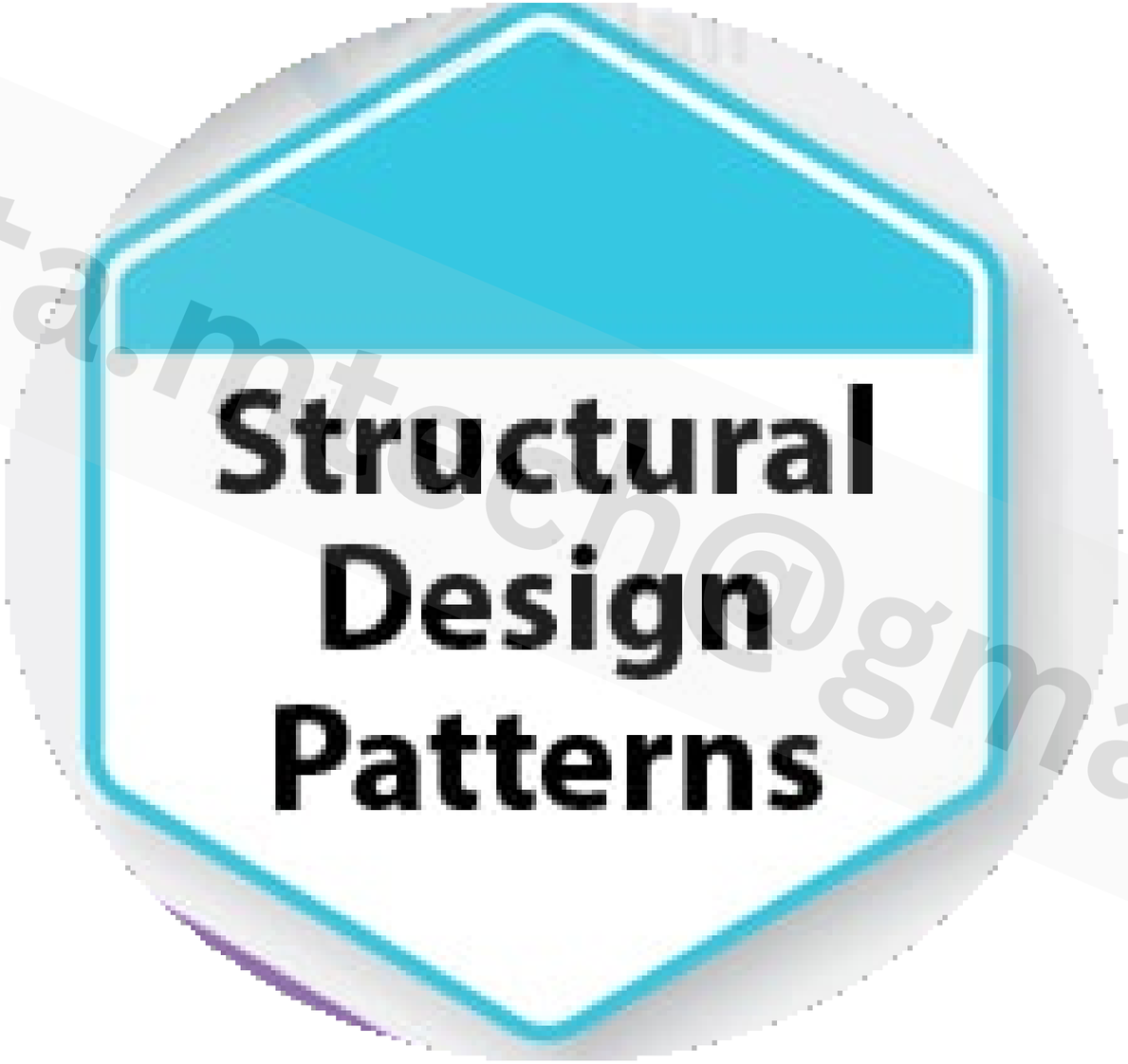
java.lang.StringBuilder#append() (unsynchronized)
java.lang.StringBuffer#append() (synchronized)

LARGE NUMBER OF VARIABLES

*there may be a **large amount** of parameters in a constructor*

*because there may be several **instance variables** in a given class*

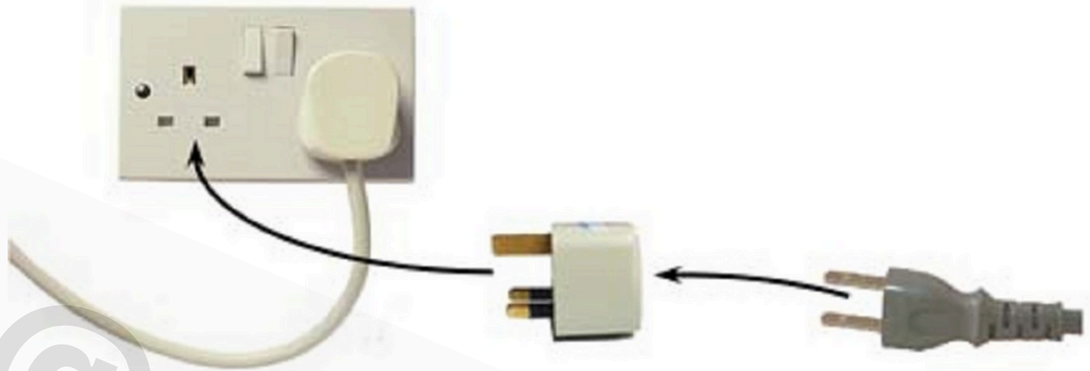
EASY TO CONFUSE THE PARAMETE



Busy Coder Academy

Adapter Pattern

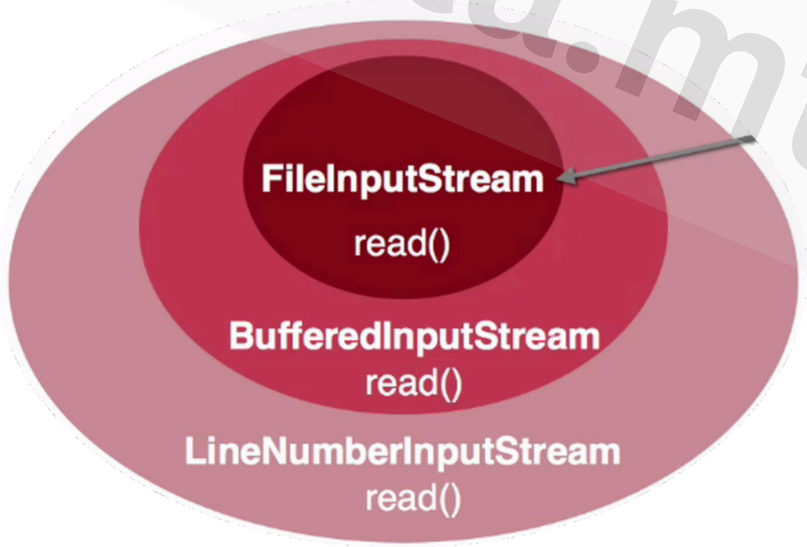
The Adapter pattern is used so that two unrelated interfaces can work together.



- `java.util.Arrays#asList()`
- `java.util.Collections#list()`
- `java.util.Collections#enumeration()`
- `java.io.InputStreamReader(InputStream)` (returns a `Reader`)
- `java.io.OutputStreamWriter(OutputStream)` (returns a `Writer`)

Decorator design pattern

Series of wrapper class that define functionality, In the Decorator pattern, a decorator object is wrapped around the original object.



Adding behaviour statically or dynamically

Extending functionality without effecting the behaviour of other objects.

Adhering to Open for extension, closed for modification.

All subclasses of `java.io.InputStream`, `OutputStream`, `Reader` and `Writer` have a constructor taking an instance of same type.

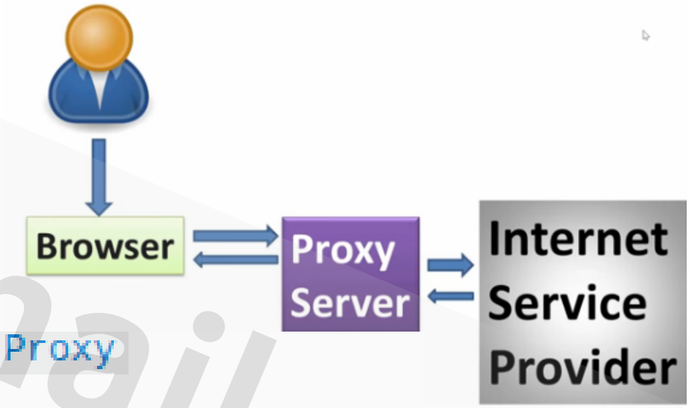
`java.util.Collections`, the `checkedXXX()`, `synchronizedXXX()` and `unmodifiableXXX()` methods.

`javax.servlet.http.HttpServletRequestWrapper` and `HttpServletResponseWrapper`

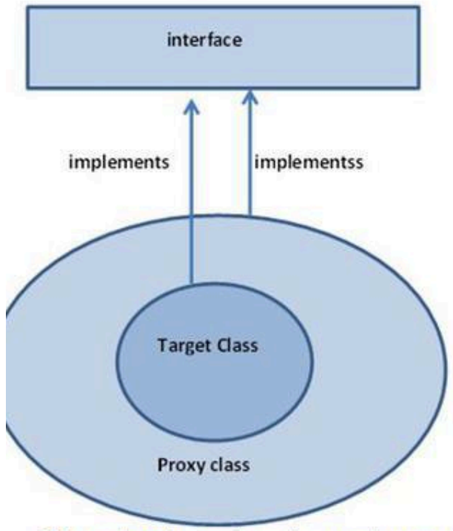
`javax.swing.JScrollPane`

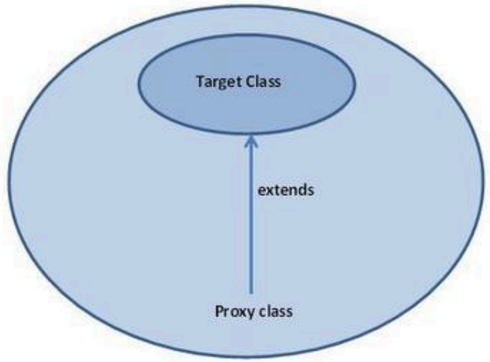
Proxy design Pattern

Proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes.



In Spring Framework AOP is implemented by creating proxy object for your service.





CGLIB proxy

`java.lang.reflect.Proxy`

`java.rmi.*`

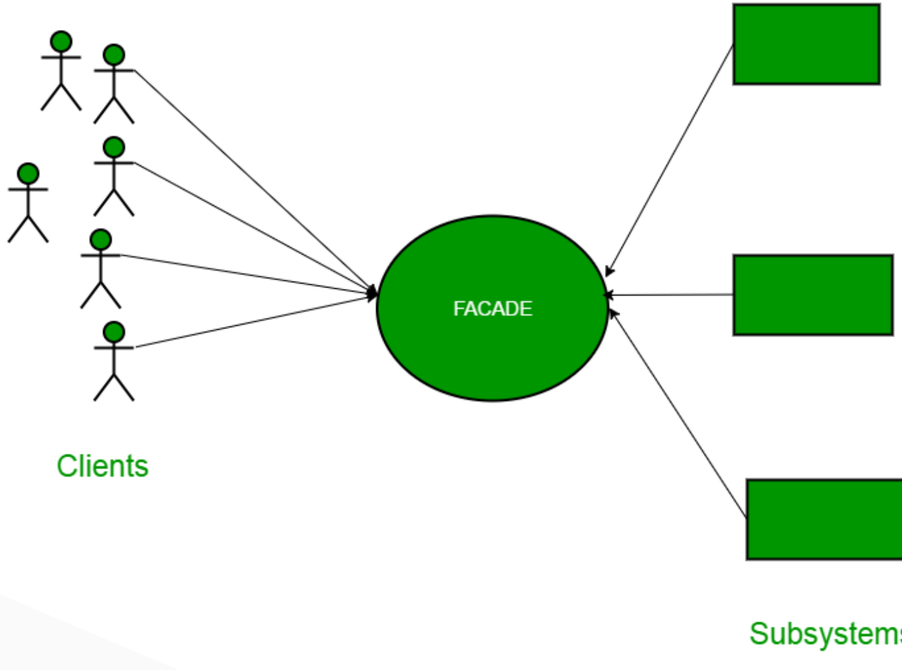
`javax.ejb.EJB` (explanation here)

`javax.inject.Inject` (explanation here)

`javax.persistence.PersistenceContext`

Busy Coder Academy

Facade Pattern



The **facade pattern** (also spelled as **façade**) is a software-**design pattern** commonly used with object-oriented programming. The name is an analogy to an architectural **façade**. A **facade** is an object that provides a simplified interface to a larger body of code, such as a class library.

- `javax.faces.context.FacesContext`, it internally uses among others the abstract/interface types `LifeCycle`, `ViewHandler`, `NavigationHandler` and many more without that the enduser has to worry about it (which are however overrideable by injection).
- `javax.faces.context.ExternalContext`, which internally uses `ServletContext`, `HttpSession`, `HttpServletRequest`, `HttpServletResponse`, etc.

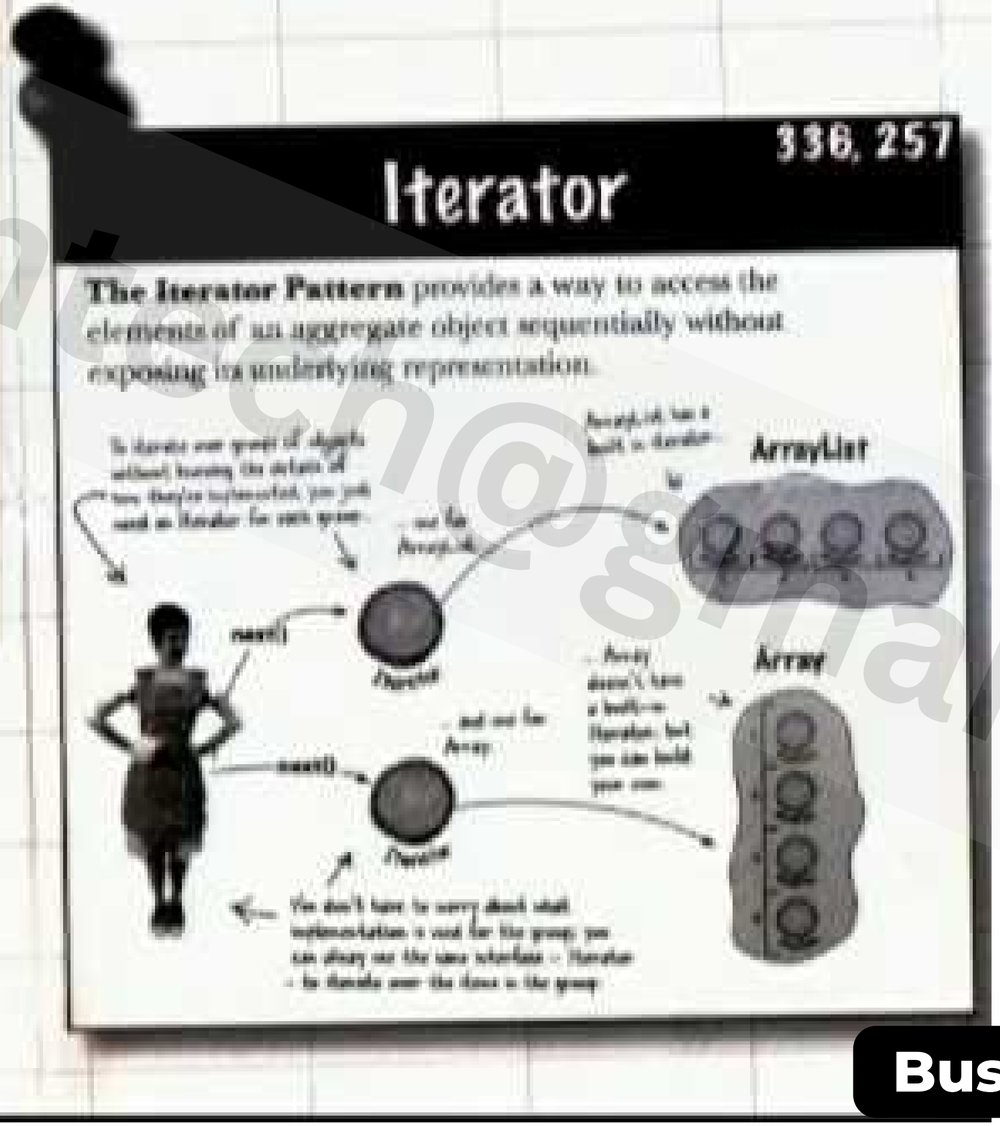
Busy Coder Academy

Iterator Pattern

"The **Iterator** Pattern provides a way to access the elements of an aggregate object sequentially without exposing the underlying representation"

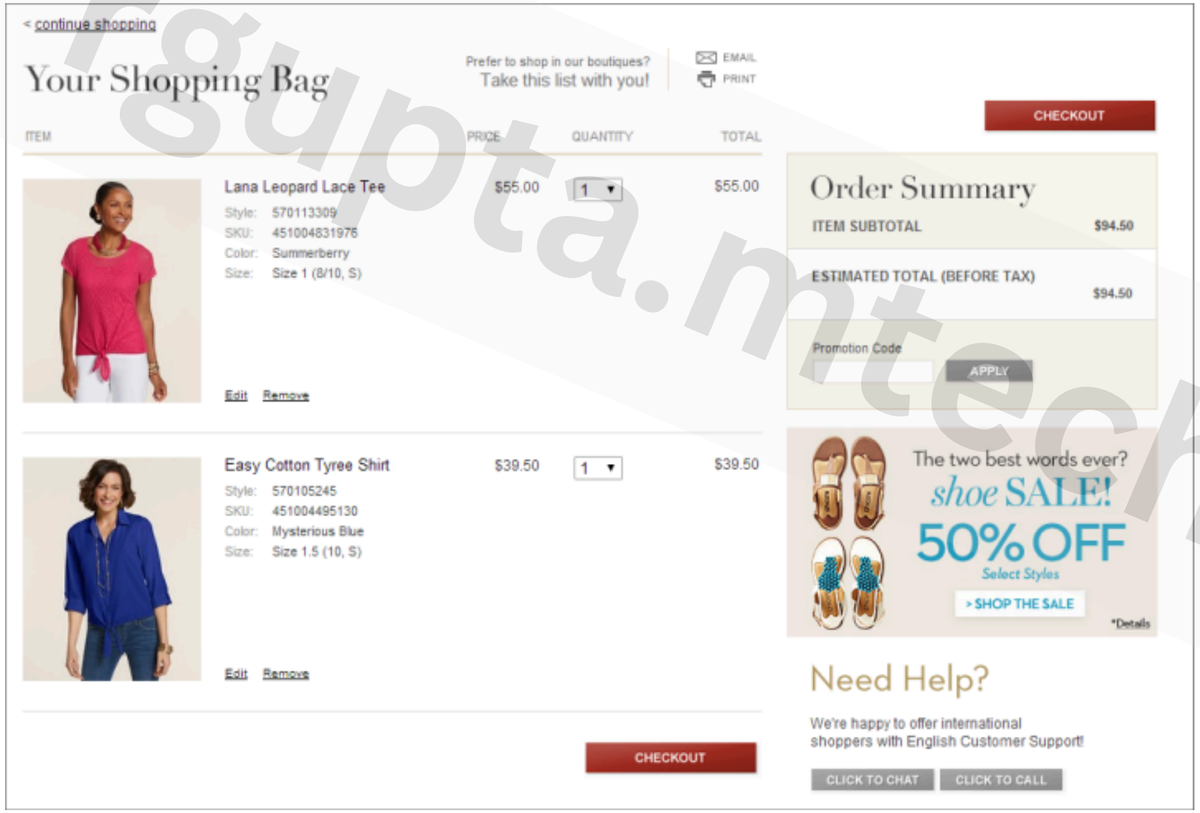
Head First Design Patterns Poster
O'Reilly,
ISBN 0-596-10214-3

Turing.com.br



Busy Coder Academy

Strategy pattern /policyPattern



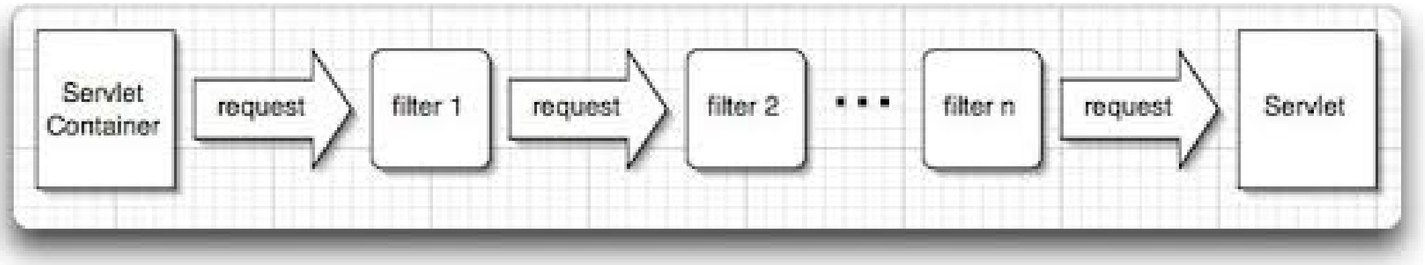
Strategy - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- `java.util.Comparator#compare()`, executed by among others `Collections#sort()`.
- `javax.servlet.http.HttpServlet`, the `service()` and all `doXXX()` methods take `HttpServletRequest` and `HttpServletResponse` and the implementor has to process them (and not to get hold of them as instance variables!).

Chain of Responsibility Pattern

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them.

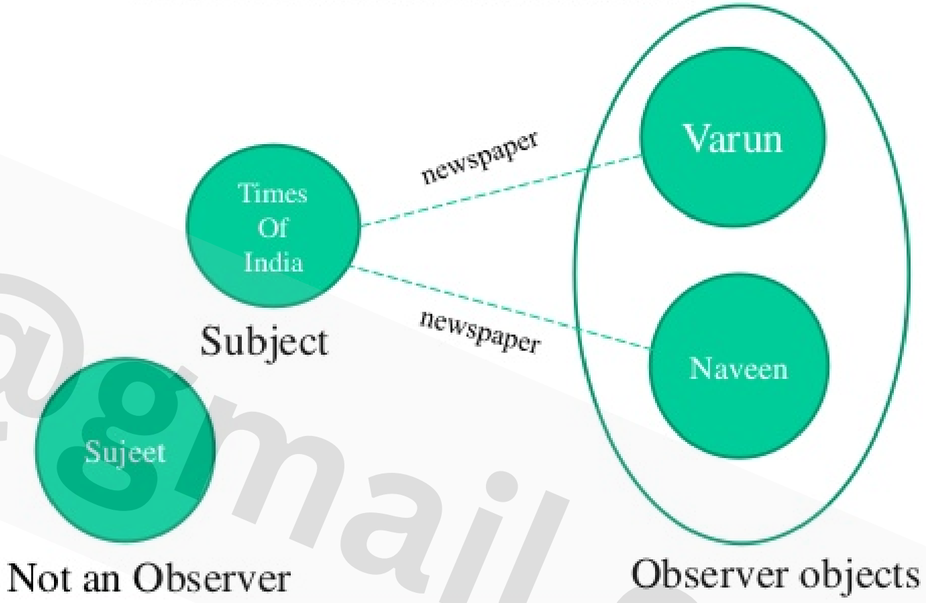
Servlet Filter, Spring Security FilterChain



Observer Design Pattern

Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change.

- Publisher + Subscribers = observer pattern
- In observer pattern publisher is called the subject and subscriber is called the observer



- `java.util.Observer` / `java.util.Observable` (rarely used in real world though)
- All implementations of `java.util.EventListener` (practically all over Swing thus)
- `javax.servlet.http.HttpSessionBindingListener`
- `javax.servlet.http.HttpSessionAttributeListener`
- `javax.faces.event.PhaseListener`

Template Design Pattern

Template Method - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses / Template Method lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure.

Foundation

Pillars

Walls

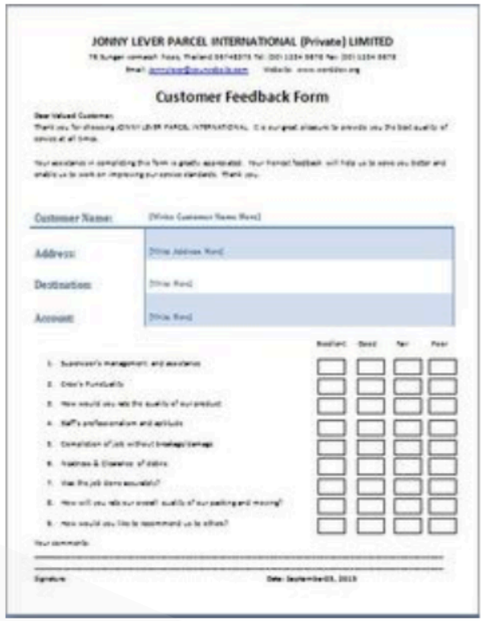
Windows

Template Method [Build a House]

1) Building foundation
2) Building pillars
3) Building walls
4) Building windows

Concrete House

Wooden House



spring JDBC

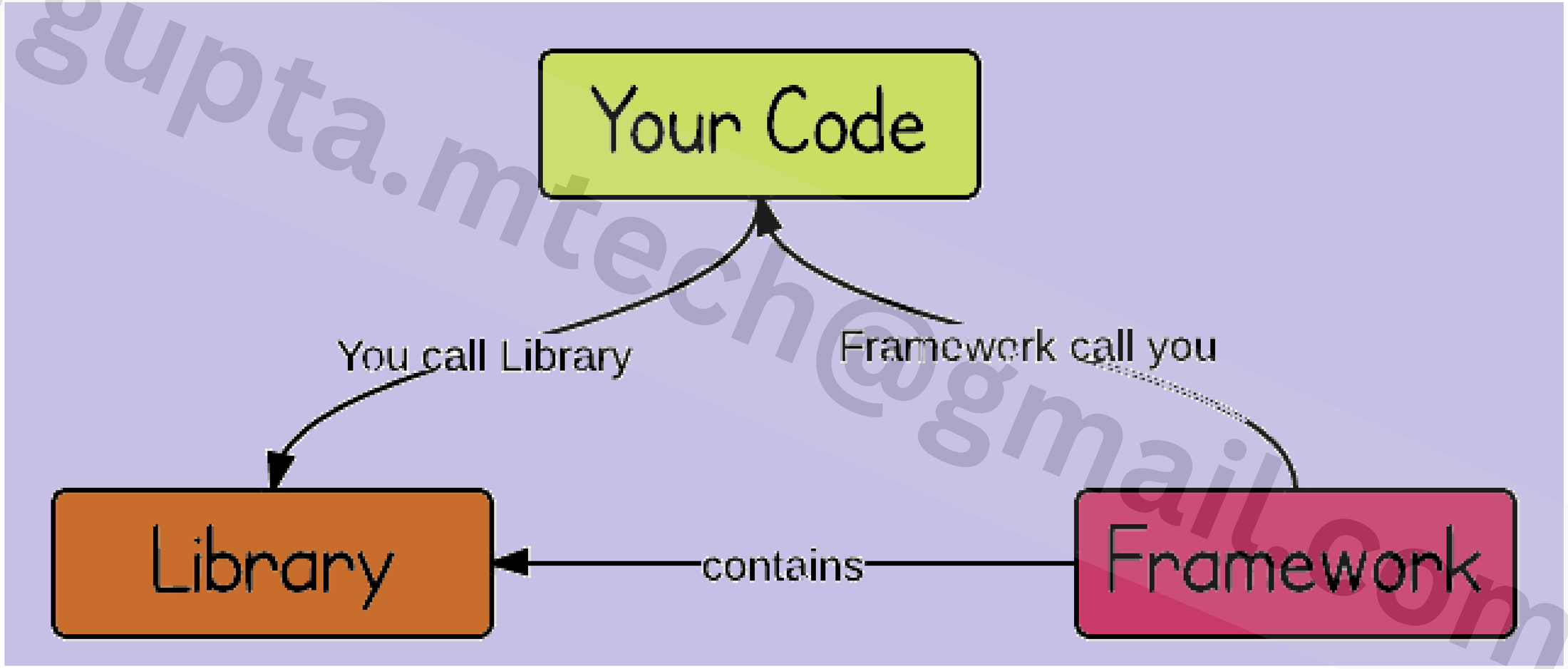
Busy Coder Academy

Pattern vs Framework

- Pattern is a set of guidelines on how to architect the application
- When we implement a pattern we need to have some classes and libraries
- **Thus, pattern is the way you can architect your application.**

- Framework helps us to follow a particular pattern when we are building a web application
- These prebuilt classes and libraries are provided by the MVC framework.
- **Framework provides foundation classes and libraries.**

Library vs Framework



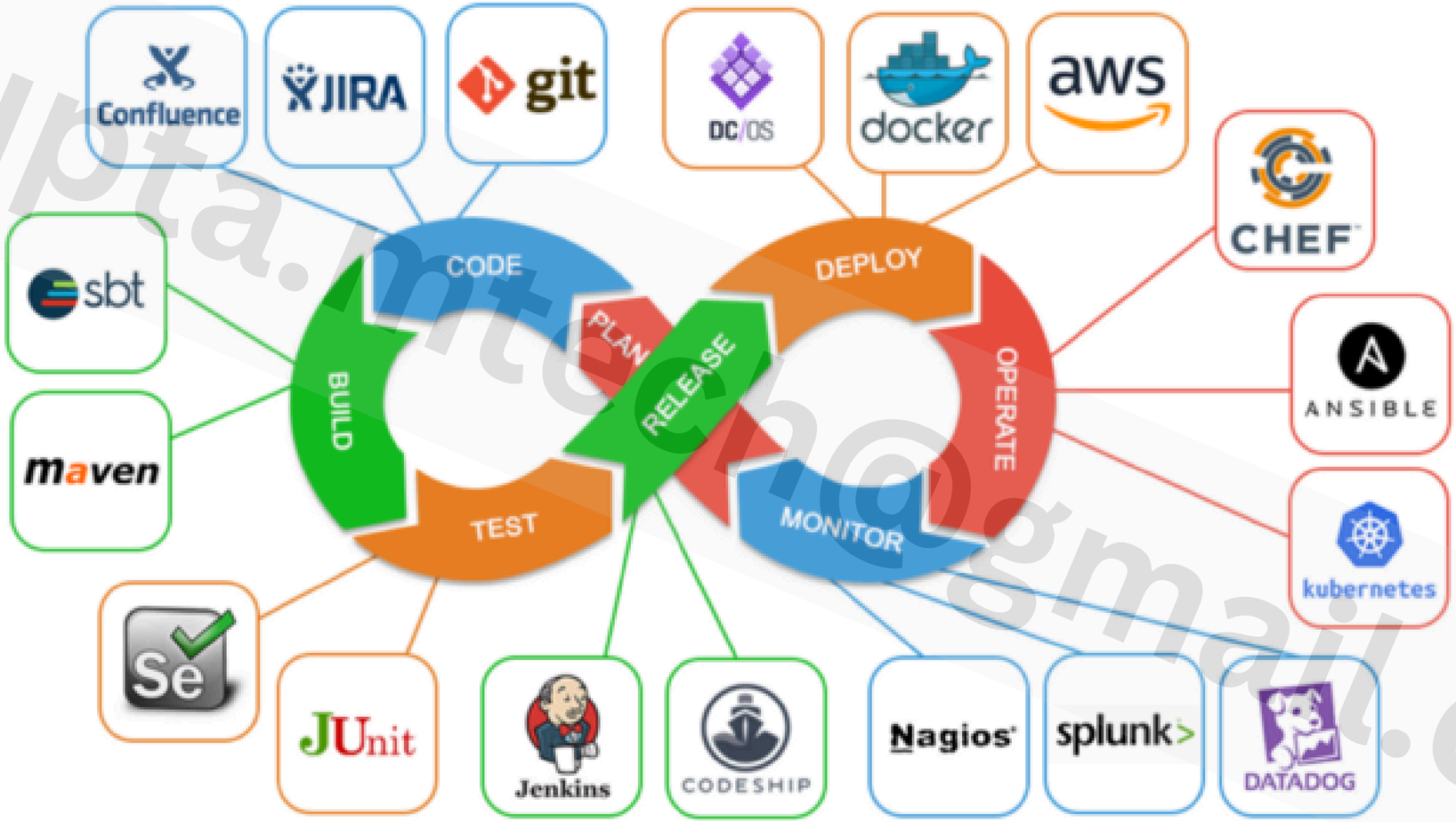
Day-4
Git, Maven, JDBC
Coding standard

rgupta.mtech@gmail.com

Day-4
Session -1
Git, Maven

rgupta.mtech@gmail.com

Introduction to DevOps



Devops is combination of development and operations.
The main objective of devops is to implement collaboration between development and operations teams.

Maven Build tool

Build management

- It is the process of compiling and assembling a software system
- Build automation is the act of scripting or automating a wide variety of tasks
  - Compiling source code
  - Packing binaries
  - Running automated tests
  - Deploying to production system
  - Creating documentation

Advantage of Build Automation

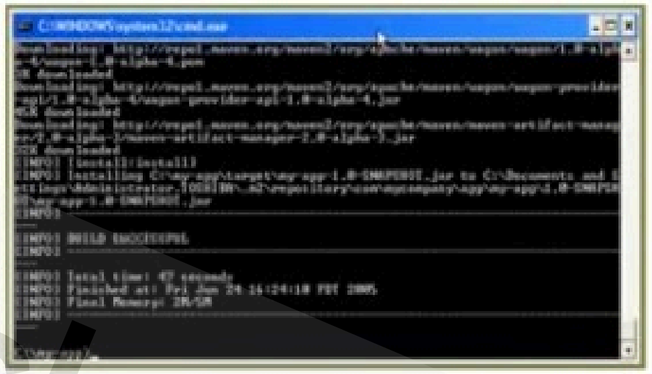
- The advantages of build automation to software development projects include
- Eliminate redundant tasks
- Accelerate the compile and link processing
- Improve product quality
- Minimize "bad builds"
- Eliminate dependencies on key personnel
- Have history of builds and releases in order to investigate issues
- Save time and money - because of the reasons listed above

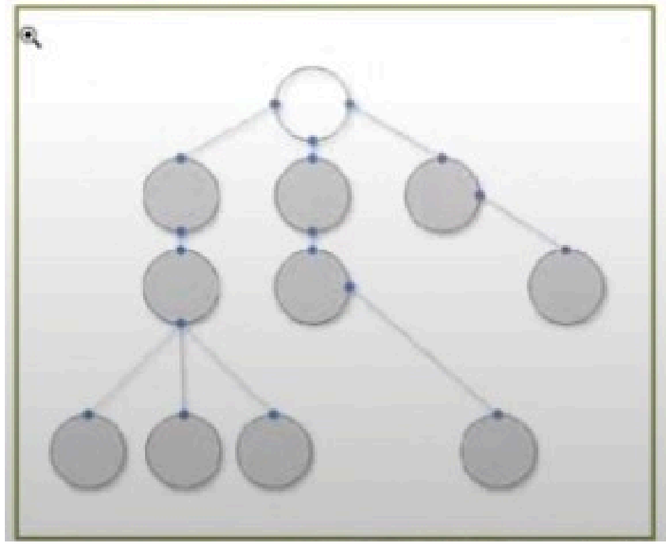
Why Maven?

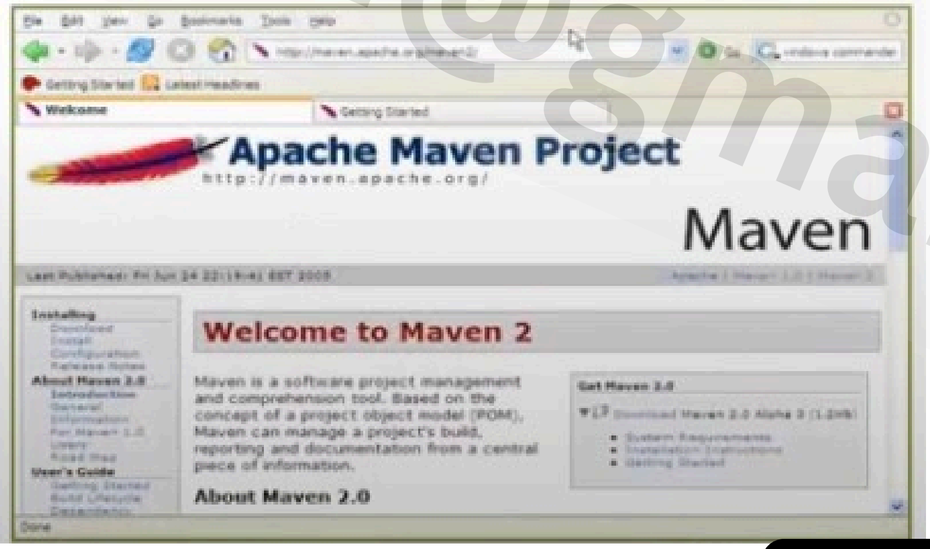
- Maven is more than just Build Tool
- Maven was built considering certain objectives
- Maven Provides:
  - Easy Build Process
  - Uniform Build System
  - Quality Project Information
  - Guidelines for Best Practices Development
- Achieved Characteristics:
  - Visibility
  - Reusability
  - Maintainability
  - Comprehensibility “Accumulator of Knowledge”

Why Maven?

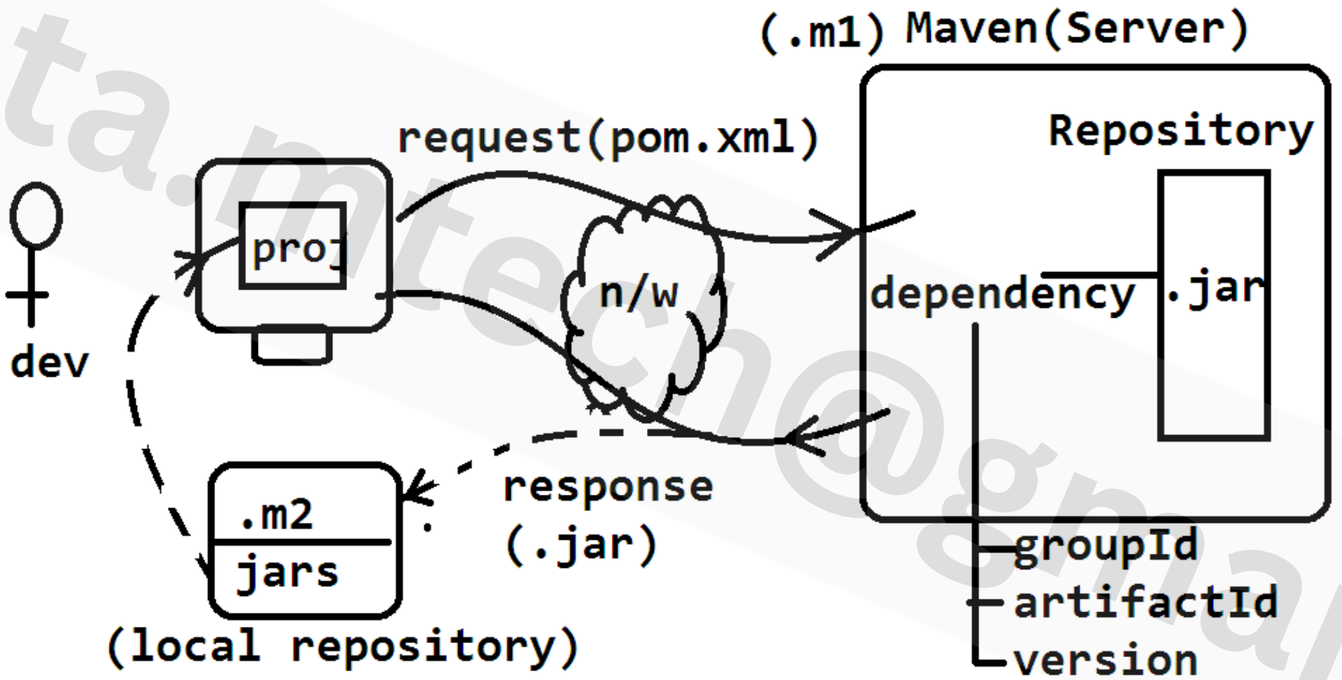
- Build-Tool
- Dependency Management Tool
- Documentation Tool







Maven Architecture



Maven Build life cycle

- A Maven build follow a lifecycle
- Default lifecycle
  - generate-sources/generate-resources
  - compile
  - test
  - package
  - Install
  - deploy
- There is also a Clean, Site lifecycle

rgupta.mtech@gmail.com

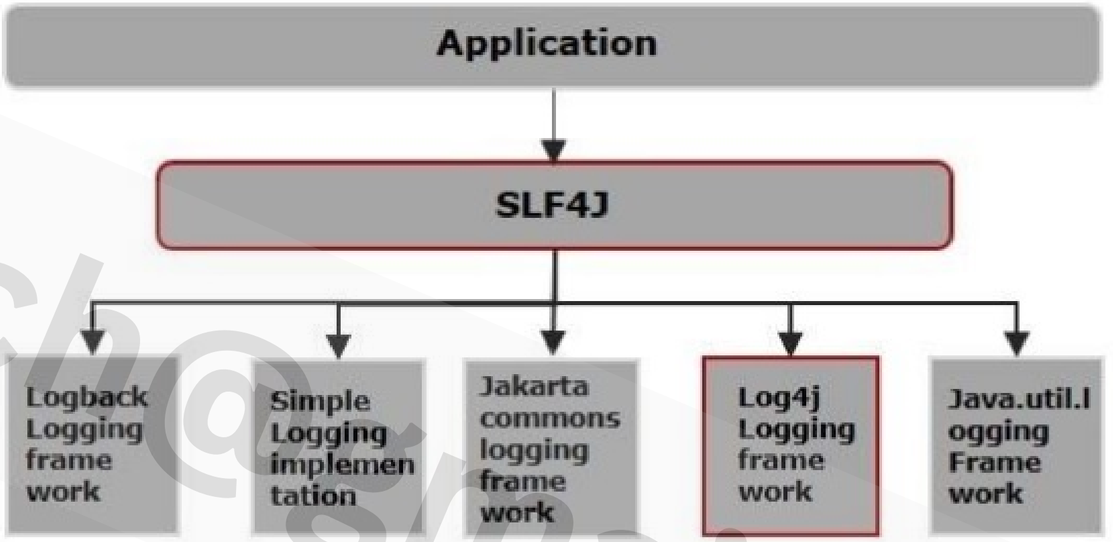
Intrdocution to Log4j2

rgupta.mtech@gmail.com

What is logging?

- logging is essential for debugging and for maintaing our application
- We must know what is going in our application, specially when error come SOP and printing exception message is not good
- Writing system.out.println(" ");//Should not be used for debugging Why?
  - As it is very hard to remove those unnessary Sop once coding is done
  - It may produce serious problem in production enveronment headach for admin peoples
  - Real advantage of logging is that it can be enable/disable and debugging messages can be directed to the file

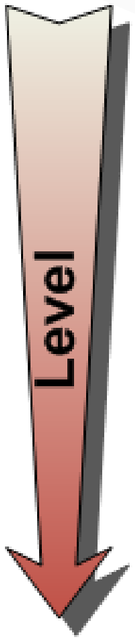
Logging framewrok?



Log 4j
log back
Commons logging
java.util.logging

most commonly used one is log4j
we should not fix ourself with any one specific logging framework as we have to change as required....
go for facade ...use Simple Logging Facade for Java

Logging levels



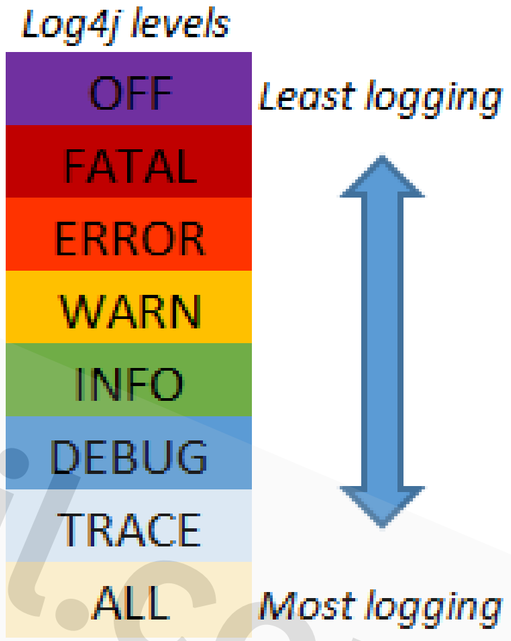
DEBUG fine-grained informational events that are most useful to debug an application

INFO informational messages that highlight the progress of the application at coarse-grained level

WARN potentially harmful situations

ERROR error events that might still allow the application to continue running

FATAL very severe error events that will presumably lead the application to abort



Day-4
Session -2
JDBC

rgupta.mtech@gmail.com

Introduction to JDBC

- **JDBC is a specification of Sun Microsystems for database connectivity,**
- **Vender is going to implement it.**
- **JDBC API has two major packages**
- **java.sql**
- **javax.sql**
- **JDBC architecture consist of different layers and drivers which are capable**
- **of working with any database.**
    -

rgupta.mtech@gmail.com

Driver & DriverManager

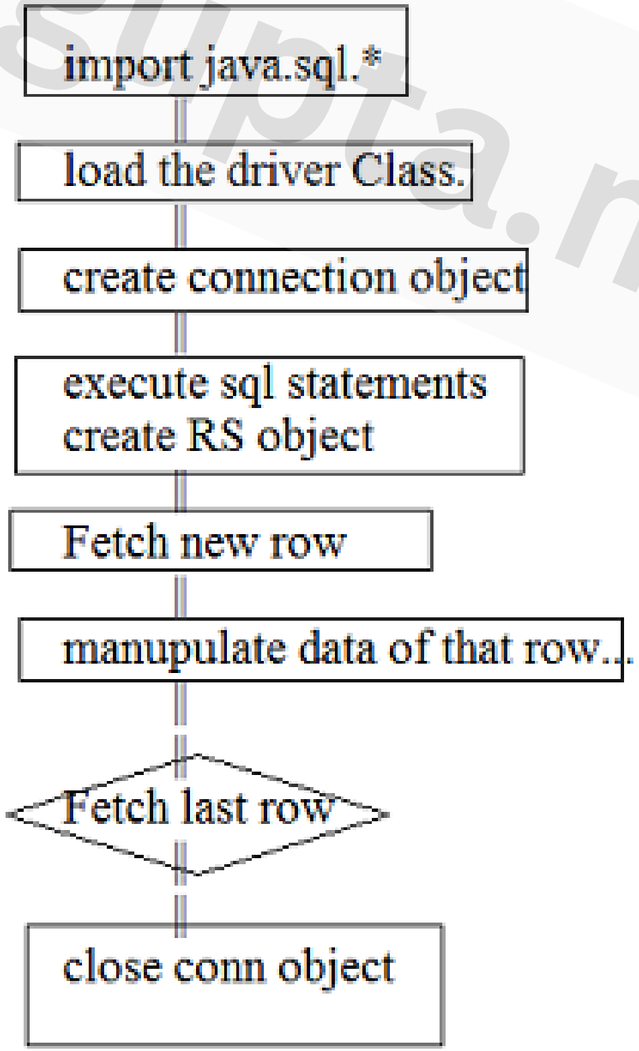
- **Driver:**
  - **Ensure consistent and uniform access to any database**
  - **Driver act as a translator bw client request and database access**
- **Driver Manager**
  - **Class that manages establishment of connection**
  - **Driver Manager need to told which JDBC driver it should try to connect**

4. Thin driver / Pure java driver / type 4

- Sun provides specification ie JDBC and vendors provides implementation
- Why vendor provides implementation?

  Oracle is first vendor to provide implementation.

JDBC Hello World



- import java.sql.*
- load the driver Class
- create connection object
- execute sql statements
- create RS object
- Fetch new row
- manipulate data of that row...
- Fetch last row?
- close conn object

CRUD Operations

```
stmt.executeUpdate("Create Table dept (deptid varchar(20) NOT NULL,
name varchar(45))");
System.out.println("Table Created");
```

```
pgsql

stmt=con.createStatement();
int i=stmt.executeUpdate(
"insert into dept(deptid,deptName)values(234,'foo')");
System.out.println("no of record inserted:"+i);
```

```
i=stmt.executeUpdate("update dept set deptName='Sales'");
System.out.println(i + " Recored(s) Updated");
```

```
pgsql

i=stmt.executeUpdate("delete from dept");
System.out.println(i + " Recored(s) Deleted");
```


rgupta.mtech@gmail.com

Prepared statements

- **Refer to data values in the query string using placeholders**
- **Then tell JDBC to bind data values to the placeholder and it will handles any special char accordingly**
- **Being a subclass of Statement, PreparedStatement inherits all the functionality of Statement.**
- **The three methods are execute(), executeQuery(), and executeUpdate()**
- **No SQL injection problem**
  -

```
stmt=con.prepareStatement(
"insert into dept(deptid,deptName)values(?,?)");
stmt.setString(1, "125");
stmt.setString(2, "tarun");
int i=stmt.executeUpdate();
```

Callable Statement

- **Sometimes the it is not possible to retrieve the whole manipulated data with a single query from database.**
- **So many queries clubbed together to form a single block.**
- **This block is known as procedures in database.**
- **So if you want to call these blocks from your java programs**
- **you need to use a special statement i.e Callable Statement.**
  -

```
CREATE PROCEDURE Test(IN num1 INT, IN num2 INT ,OUT param1 INT)
BEGIN
set param1 := num1 + num2;
END
//
DELIMITER ;
CALL test(3,6,@a);
select @a;
```

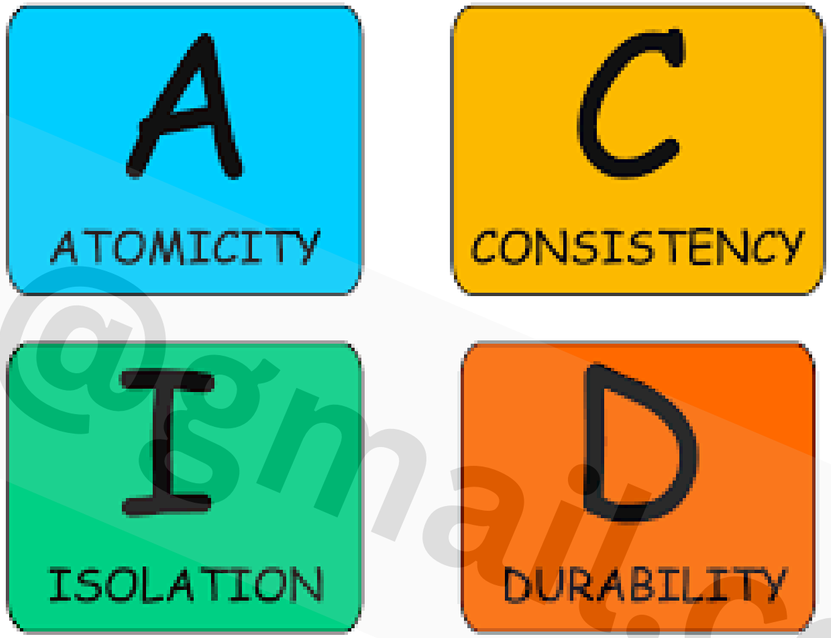
ResultSetMetaData & DatabaseMetaData

- **A Connection's database is able to provide schema information describing its tables,**
- **its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on**
- **This information is made available through a DatabaseMetaData object.**
  -

```
public static void printRS(ResultSet rs) throws SQLException
{
    ResultSetMetaData md = rs.getMetaData();
    int nCols = md.getColumnCount();

    for(int i=1; i < nCols; ++i)
        System.out.print(md.getColumnName(i)+",");

    while(rs.next())
    {
        for(int i=1; i < nCols; ++i)
            System.out.print(rs.getString(i)+",");
        System.out.println(rs.getString(nCols));
    }
}
```

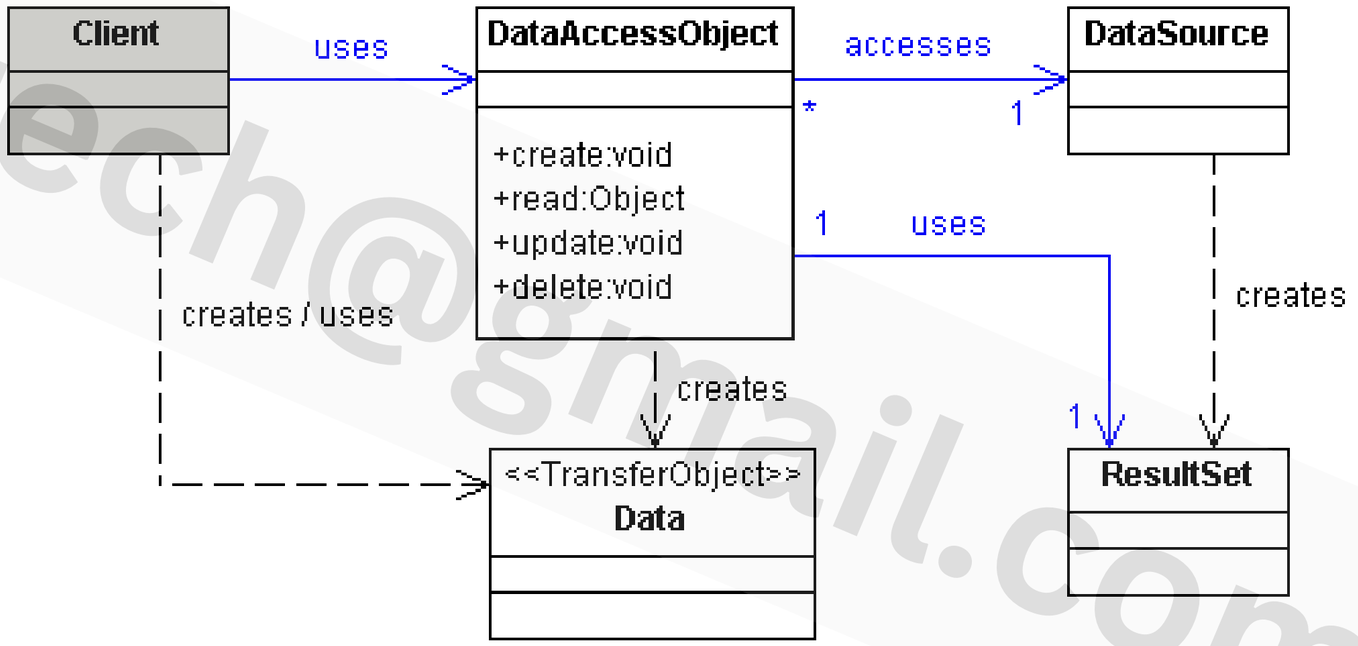
Transaction Management

- **Transaction**
  - **logical unit of related work**
  - **ACID properties**
- **By default the connection commits all the changes once it is done with the single query.**
- **This can be stopped by calling setAutoCommit(false) on connection object.**
- **Rollback() method gets called for withdrawing all the changes done by the queries in the database.**
  -



Data Access Object (DAO) Pattern

- **DAO Pattern Roles**
- **Business Logic**
- **Data Access Object (DAO)**
- **Business Object**
- **Data Source**
  -



Day 5

- Java 8 Introduction
- Stream API
- Optional
- Date and Time API

Java 8 Stream Processing

=> Introduction to java 8, Why i should care for it?
=> functional interface
=> Lambda expressions
=> diff bw ann inner class vs lambda expression
=> Passing code with behavior parameterization
=> introduction to stream processing
=> functional interfaces defined in Java 8
=> Case study stream processing
=> Parallel data processing and performance

Why i sould care about Java 8?

What is offer?

Why i sould care about Java 8?

Classical threads vs JUC

Java 7 fork and join

Java 8 parallel processing declarative data processing

SQL vs Java

Optimization should be Job of JVM or programmer?

Concurrency vs parallel processing?

rgupta.mtech@gmail.com

Java-style functional programming

JSR 335 Lambda Expression Closure
JEP 107 :Bulk data operation for collections forEach, filter, map,reduce

Java DNA

Structured ==> Reflective=> Object Oriented =>Functional => Imperative=>Concurrent => Generic

What Java is now?

".....Is a blend of imperative and object oriented programming enhanced with functional flavors"

Methods vs. Functions

	Method	Pure Function
Mutation	A method mutates , i.e., it modifies data and produces other changes and side effects.	A pure function does not mutat it just inspects data and produces results in form of new data.
How vs. What	Methods describe how things are done.	Pure functions describe what is done rather than how it is done
Invocation Order	Methods are used imperatively order of invocation matters.	Pure functions are used declaratively

Java 8 Functional interface

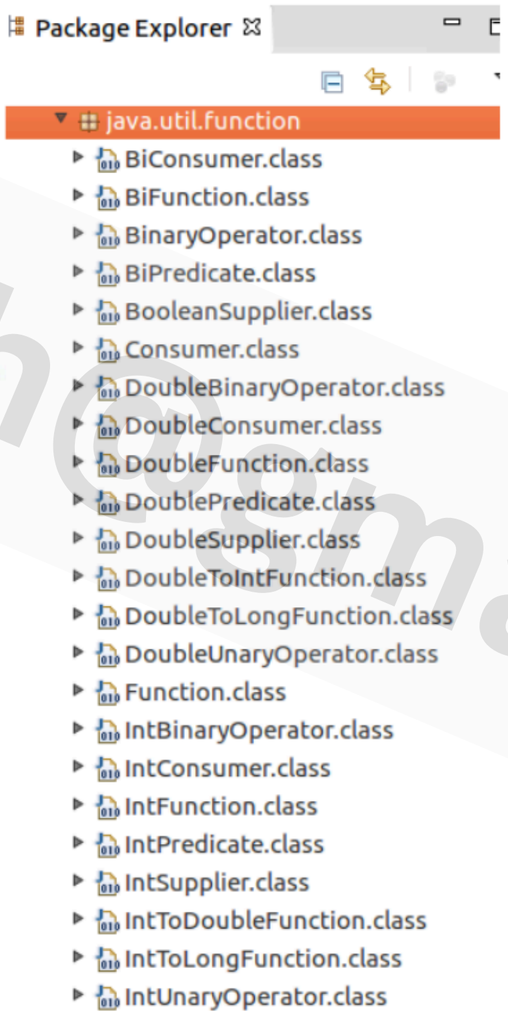
- New term of Java 8
- A functional interface is an interface with only one *abstract* method.

```
public interface Runnable {

   run();
};
```

```
public interface Comparator<T> {

   int compare(T t1, T t2);
};
```

Functional Interface

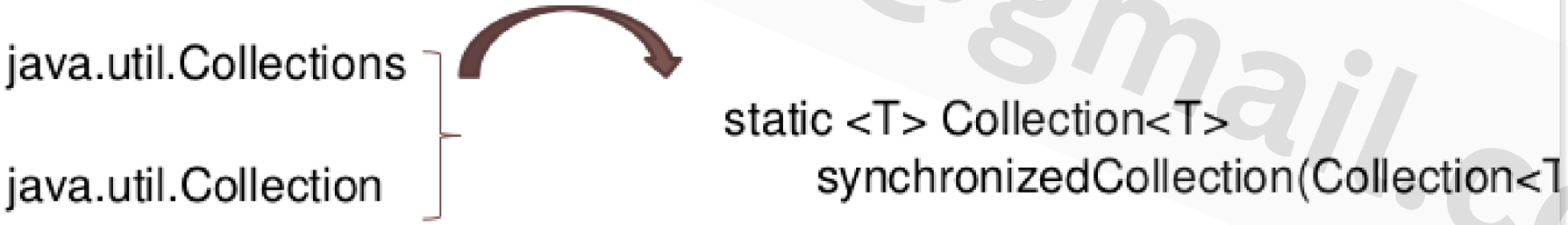
- Brand new java.util.function package.
- Rich set of function interfaces.
- 4 categories:
  - Supplier
  - Consumer
  - Predicate
  - Function



Interface: Default and static methods

- Extends interface declarations with two new concepts:
  - Default methods
  - Static methods
- Advantages:

- No longer need to provide your own companion utility classes. Instead, you can place static methods in the appropriate interfaces



- Adding methods to an interface without breaking the existing implementation

Lambda Expression?

First Class Functions

- **Old style**

```
Arrays.sort(testStrings, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return(s1.length() - s2.length());
    }
});
```

- **New style**

```
Arrays.sort(testStrings,
        (s1, s2) -> { return(s1.length() – s2.length()); });
```

Lambda Expression?

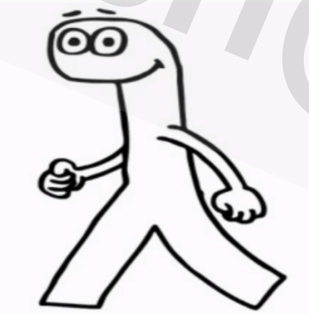
Lambda Expression vs inner classes?

Examples?

When to use lambda expression?

Performance difference?

First Class Functions



Lambda expression
==
anonymous method call
Vs Inner class

rgupta.mtech@gmail.com

Lambda Expression?

provide a clear and concise way to represent a “one abstract method interface” (a so-called **functional interface)** using an expression

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(event -> System.out.println("Click!"));
```

A lambda is composed of three parts:

Argument list	Arrow token	Body of the method
event	→	System.out.println("Click!")

It works because the listener has only one abstract method and the compiler can infer what to do from the interface

```
package java.awt.event;

import java.util.EventListener;

public interface ActionListener extends EventListener {
    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);
}
```

Lambda sample

- We need to find the books with more than 400 pages.

```
public List getLongBooks(List books) {
    List accumulator = new ArrayList<>();
    for (Book book : books) {
        if (book.getPages() > 400) {
            accumulator.add(book);
        }
    }
    return accumulator;
}
```

- Now the requirements has changed and we also need to filter for genre of the book

```
public List getLongNonFictionBooks(List books) {
    List accumulator = new ArrayList<>();
    for (Book book : books) {
        if (book.getPages() > 400 && Genre.NON_FICTION.equals(book.getGenre())) {
            accumulator.add(book);
        }
    }
    return accumulator;
}
```

- We need a different method for every filter, while the only change is the if condition

Lambda sample

- We can use a lambda. First we define a functional interface, which is an interface with only one abstract method

```
@FunctionalInterface
public interface BookFilter {

    public boolean test(Book book);
}
```

- Then we can define a generic filter and write as many implementation we want in just one line

```
public static List lambdaFilter(List books, BookFilter bookFilter) {
    List accumulator = new ArrayList<>();
    for (Book book : books) {
        if (bookFilter.test(book)) {
            accumulator.add(book);
        }
    }
    return accumulator;
}

// one line filters
List longBooks = lambdaFilter(Setup.books, b -> b.getPages() > 400);

BookFilter nflbFilter = b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre();
List longNonFictionBooks = lambdaFilter(Setup.books, nflbFilter);
```

Functional Interface Java 8

- We don't need to write all the functional interfaces because Java 8 API defines the basic ones in *java.util.function package*

Functional interface	Descriptor	Method name
Predicate<T>	T → boolean	test()
BiPredicate<T, U>	(T, U) → boolean	test()
Consumer<T>	T → void	accept()
BiConsumer<T, U>	(T, U) → void	accept()
Supplier<T>	() → T	get()
Function<T, R>	T → R	apply()
BiFunction<T, U, R>	(T, U) → R	apply()
UnaryOperator<T>	T → T	identity()
BinaryOperator<T>	(T, T) → T	apply()

- So we did not need to write the BookFilter interface, because the Predicate interface has exactly the same descriptor

Lambda sample

So we can rewrite our code as

```
public static List lambdaFilter(List books, Predicate bookFilter) {
    List accumulator = new ArrayList<>();
    for (Book book : books) {
        if (bookFilter.test(book)) {
            accumulator.add(book);
        }
    }
    return accumulator;
}

// one line filters
List longBooks = lambdaFilter(Setup.books, b -> b.getPages() > 400);

Predicate nflbFilter = b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre();
List longNonFictionBooks = lambdaFilter(Setup.books, nflbFilter);
```

Lambdas and existing interfaces

Since in JDK there are a lot of interfaces with only one abstract method, we can use lambdas also for them:

```
// Runnable interface defines void run() method
Runnable r = () -> System.out.println("I'm running!");
r.run();

// Callable defines T call() method
Callable callable = () -> "This is a callable object";
String result = callable.call();

// Comparator defines the int compare(T t1, T t2) method
Comparator bookLengthComparator = (b1, b2) -> b1.getPages() - b2.getPages();
Comparator bookAgeComparator = (b1, b2) -> b1.getYear() - b2.getYear();
```

Method reference

- Sometimes code is more readable if we refer just to the method name instead of a lambda

Kind of method reference	Example
To a static method	Integer::parseInt
To an instance method of a class	Integer::intValue
To an instance method of an object	n::intValue
To a constructor	Integer::new

- So we can rewrite this lambda

```
Function<String, Integer> lengthCalculator = (String s) -> s.length();
```

- with a method reference

```
Function<String, Integer> lengthCalculator = String::length;
```

Comparators

- In former versions of Java, we had to write an anonymous inner class to speficy the behaviour of a Comparator

```
Collections.sort(users, new Comparator<Author>() {
    public int compare(Author a1, Author a2) {
        return a1.compareTo(a2.id);
    }
});
```

- We can use lambda for making code more readable:

```
// now sort is a oneliner!
Collections.sort(authors, (Author a1, Author a2) -> a1.compareTo(a2));
```

Streams

- The Java Collections framework relies on the concept of external iteration, as in the example below

```
for (Book book: books) {
    book.setYear = 1900;
}
```

- compared to internal iteration, like the example below

```
Books.forEach(b -> book.setYear(1900));
```

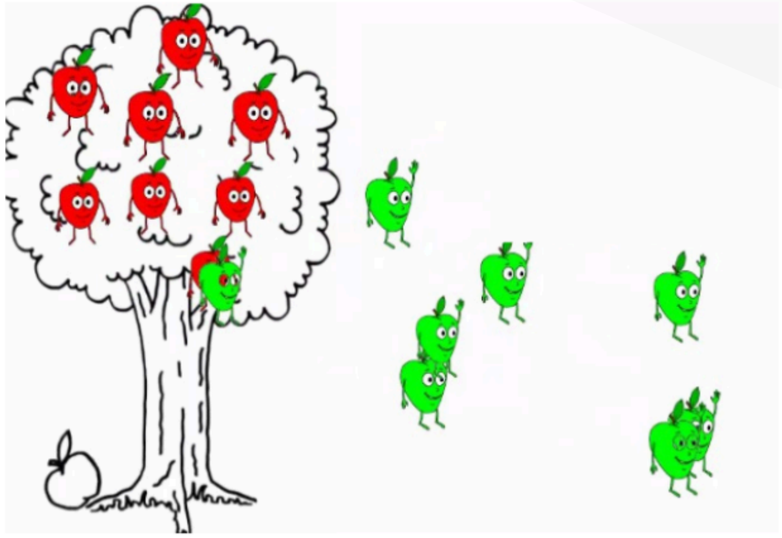
- The difference is not only in code readabilty and maintainability, is also related to performance: the runtime can optimize the internal iteration for parallelism, lazyness or reordering the data

Behavior parameterization
java 8
Light weight way to implement strategy design pattern
In Java 8?

rgupta.mtech@gmail.com

Consider Example

Green Apples, Please...



Now Heavy Ones , Please...

Behavior parameterization

```
public static List<Apple> filterGreenApples
                          (List<Apple> inventory){
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (apple.getColor.equals("green")) {
            result.add(apple);
        }
    }
    return result;
}
```

```
public static List<Apple> filterHeavyApples
                          (List<Apple> inventory){
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (apple.getWeight() > 150) {
            result.add(apple);
        }
    }
    return result;
}
```

Behavior parameterization

New Behavior

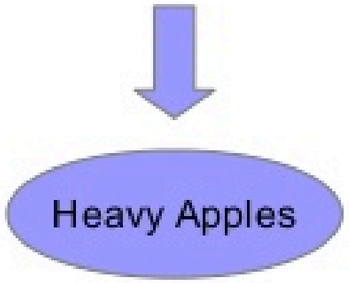
apple.getWeight > 150

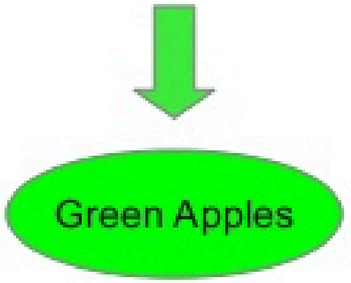
apple.getColor.equals("green")

Behavior Parameterization

```
static List<Apple> filterApples(
        List<Apple> inventory, Predicate<Apple> p) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

Output





Stream processing
An introduction

rgupta.mtech@gmail.com

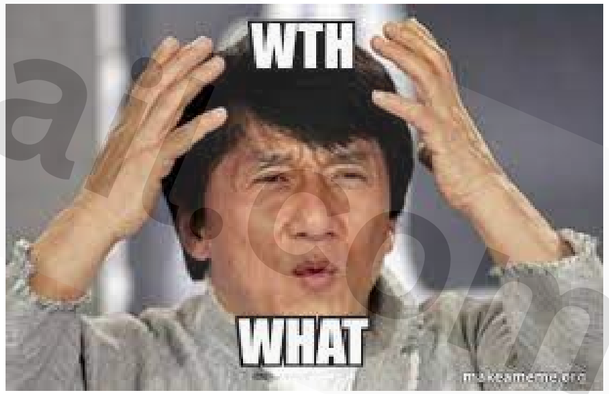
Data processing requirment

Return the names of dishes that are low in calories (<400), Sorted by number of calories

```
List<Dish> foods=new ArrayList<Dish>();
for(Dish d: foodsAll){
    if(d.getCalories()< 400)
        foods.add(d);
}
//now sorting as per calories
```

```
Collections.sort(foods, new Comparator<Dish>() {

    @Override
    public int compare(Dish o1, Dish o2) {
        return Integer.compare(o1.getCalories(), o2.getCalories());
    }
});
//now we want only names
```

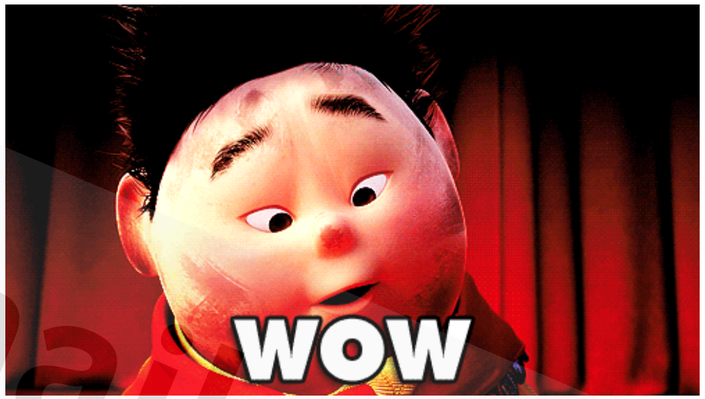
```
List<String> names=new ArrayList<String>();
for(Dish d: foods){
    names.add(d.getName());
}
```



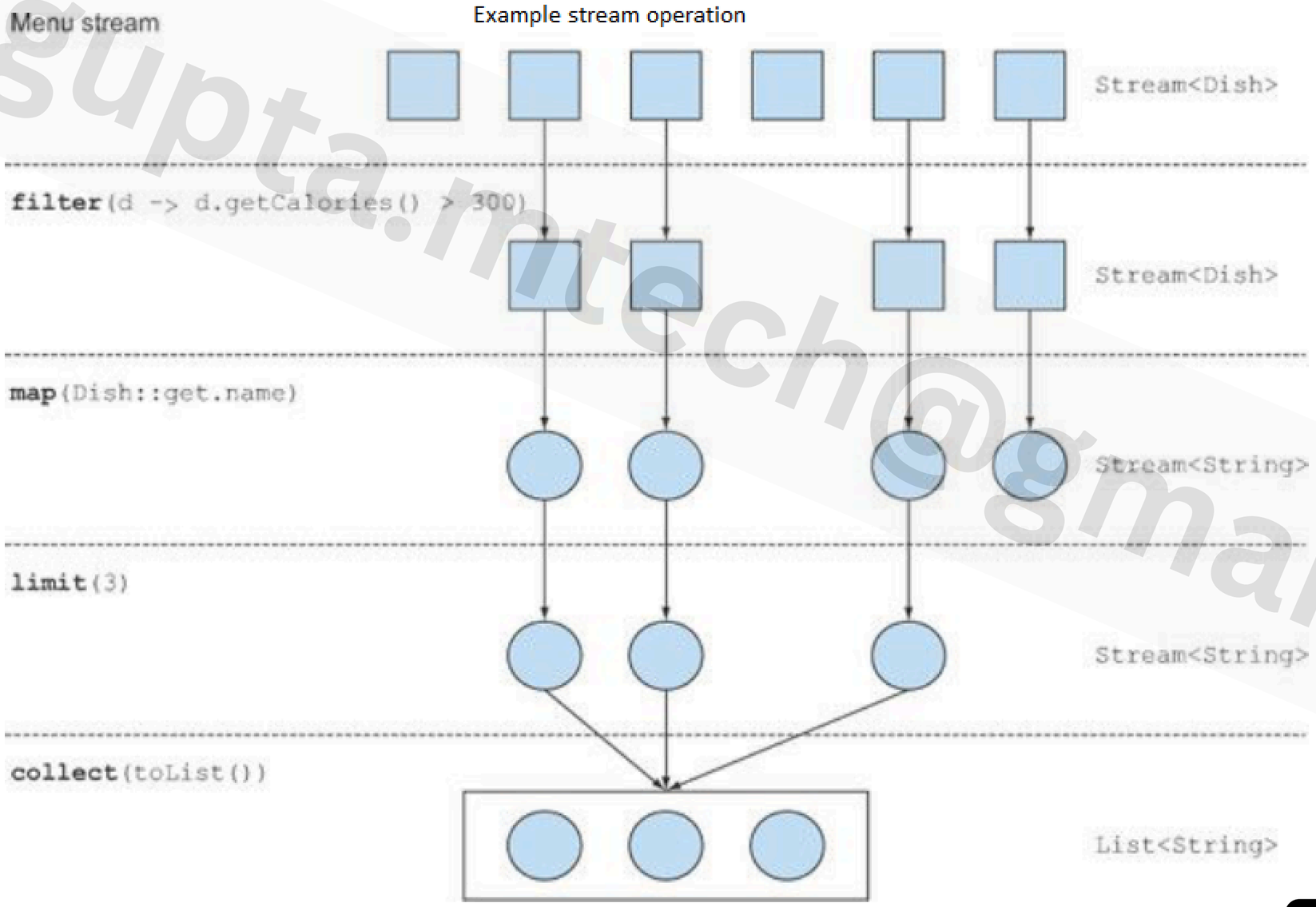
Data processing Java 8 Steams

Return the names of dishes that are low in calories (<400), Sorted by number of calories

```
List<String> selectedFoodsNames=foodsAll
        .stream()
        .filter(f-> f.getCalories()< 400)
        .sorted(Comparator.comparing(Dish::getCalories))
        .map(f-> f.getName())
        .collect(Collectors.toList());
```



Data processing Java 8 Steams

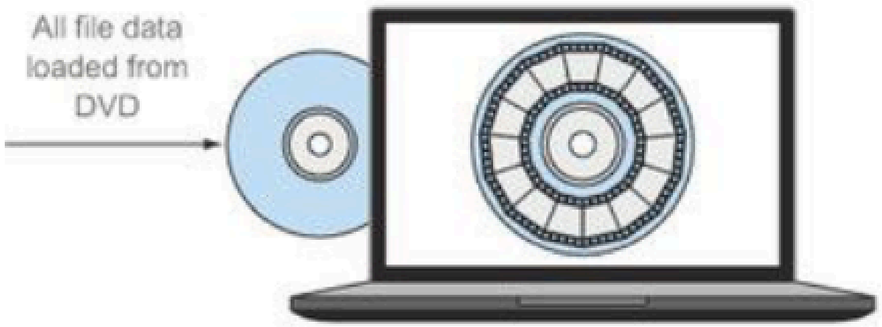


Stream vs collection?

A collection in Java 8 is like a movie stored on DVD

Eager construction means waiting for computation of ALL values

All file data loaded from DVD



Like a DVD, a collection holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection.

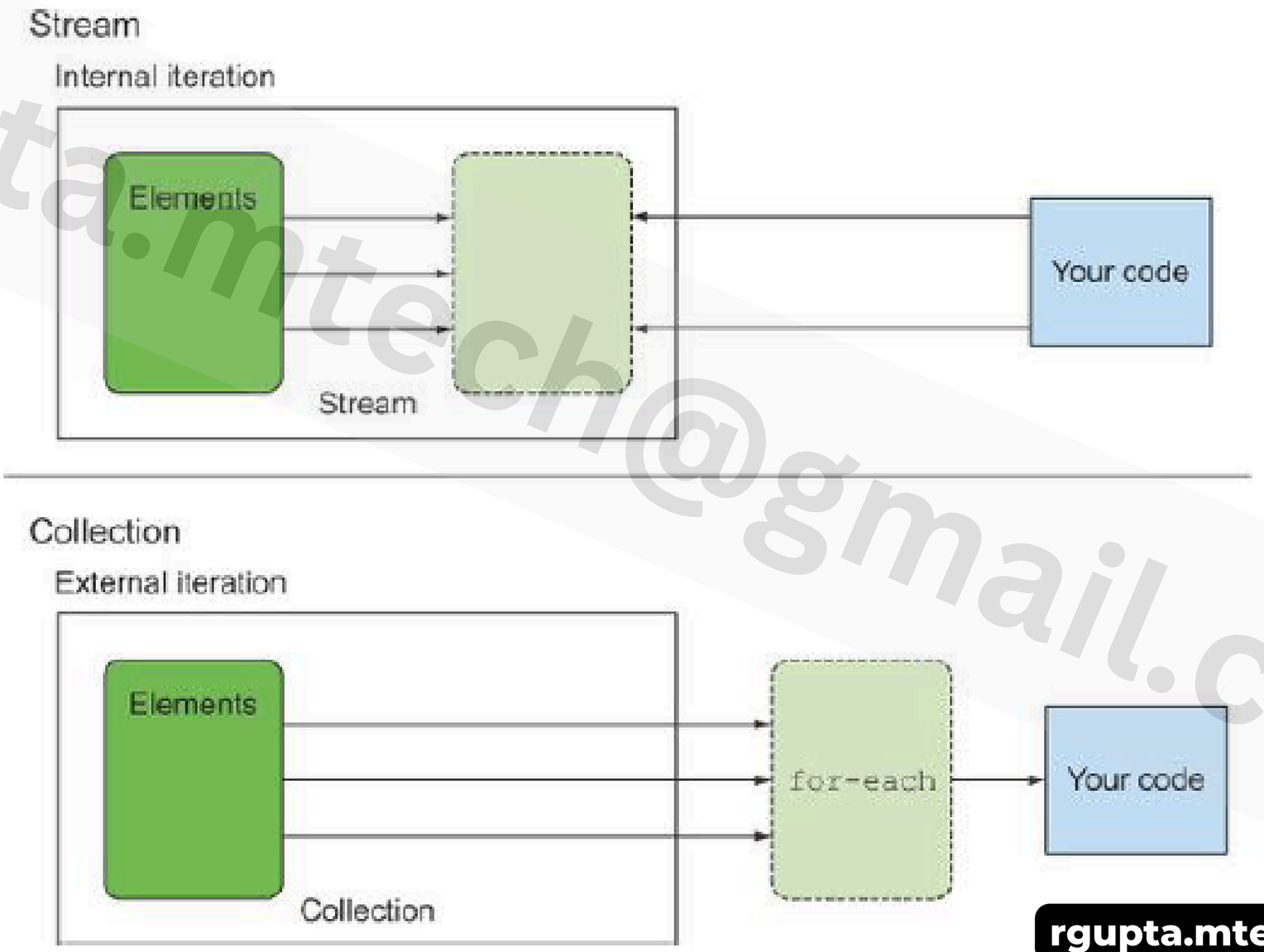
A stream in Java 8 is like a movie streamed over the internet.

Lazy construction means values are computed only as needed.

Internet

Like a streaming video, values are computed as they are needed.

Internal vs external iteration?



Intermediate operations?

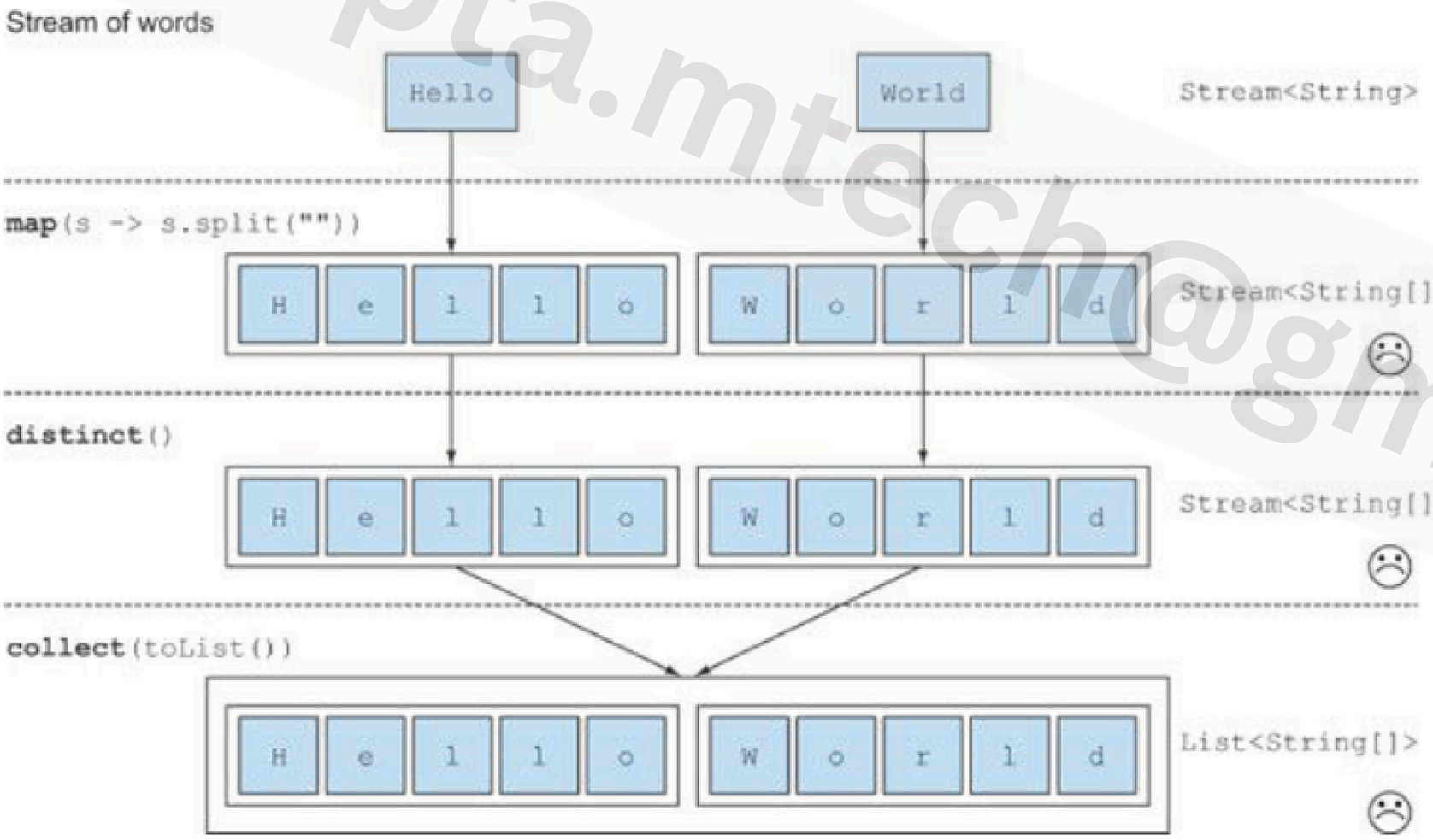
Operation	Type	Return type	Argument of the operation	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
map	Intermediate	Stream<R>	Function<T, R>	T -> R
limit	Intermediate	Stream<T>		
sorted	Intermediate	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Intermediate	Stream<T>		

Terminal operation?

Operation	Type	Purpose
forEach	Terminal	Consumes each element from a stream and applies a lambda to each of them. The operation returns void.
count	Terminal	Returns the number of elements in a stream. The operation returns a long.
collect	Terminal	Reduces the stream to create a collection such as a List, a Map, or even an Integer.

Need of flatMap?

Incorrect implementation : need of flatMap



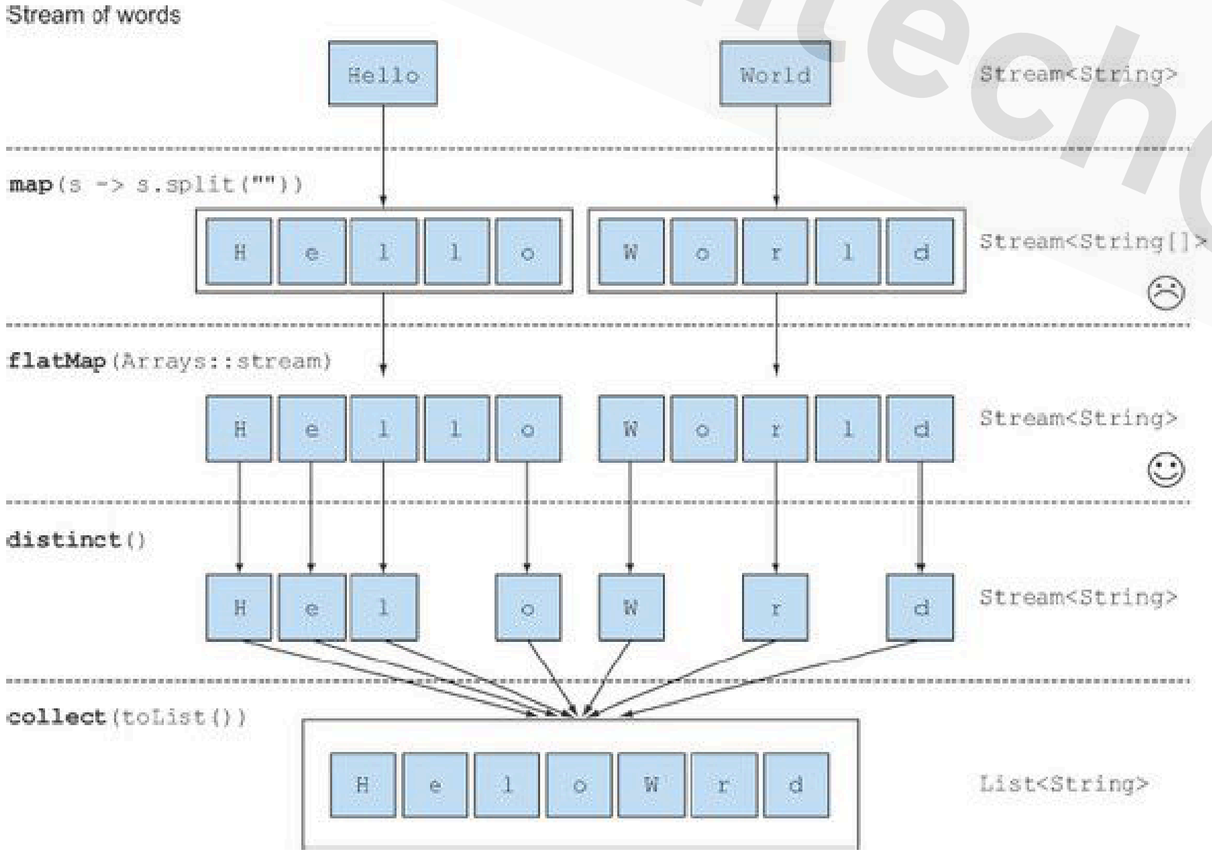

```
words.stream()
        .map(word -> word.split(""))
        .distinct()
        .collect(toList());
```

Correct Implementation using flatMap

```
List<String> uniqueCharacters =
  words.stream()
       .map(w -> w.split(""))
       .flatMap(Arrays::stream)
       .distinct()
       .collect(Collectors.toList());
```

Converts each word into an array of its individual letters

Flattens each generated stream into a single stream



Correct implementation using flatMap:

Using the flatMap method has the effect of mapping each array not wit
stream but *with the contents of that stream.* All the separate streams
that were generated when using map(Arrays::stream) get amalgamate
flattened into a single stream

Intermediate and terminal operations List

Operation	Type	Return type	Type/functional interface used	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
distinct	Intermediate (stateful-unbounded)	Stream<T>		
skip	Intermediate (stateful-bounded)	Stream<T>	long	
limit	Intermediate (stateful-bounded)	Stream<T>	long	
map	Intermediate	Stream<R>	Function<T, R>	T -> R
flatMap	Intermediate	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	Intermediate (stateful-unbounded)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	Terminal	boolean	Predicate<T>	T -> boolean
noneMatch	Terminal	boolean	Predicate<T>	T -> boolean
allMatch	Terminal	boolean	Predicate<T>	T -> boolean

Operation	Type	Return type	Type/functional interface used	Function descriptor
forEach	Terminal	void	Consumer<T>	T -> void
collect	terminal	R	Collector<T, A, R>	
reduce	Terminal (stateful-bounded)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	Terminal	long		
findAny	Terminal	Optional<T>		
findFirst	Terminal	Optional<T>		

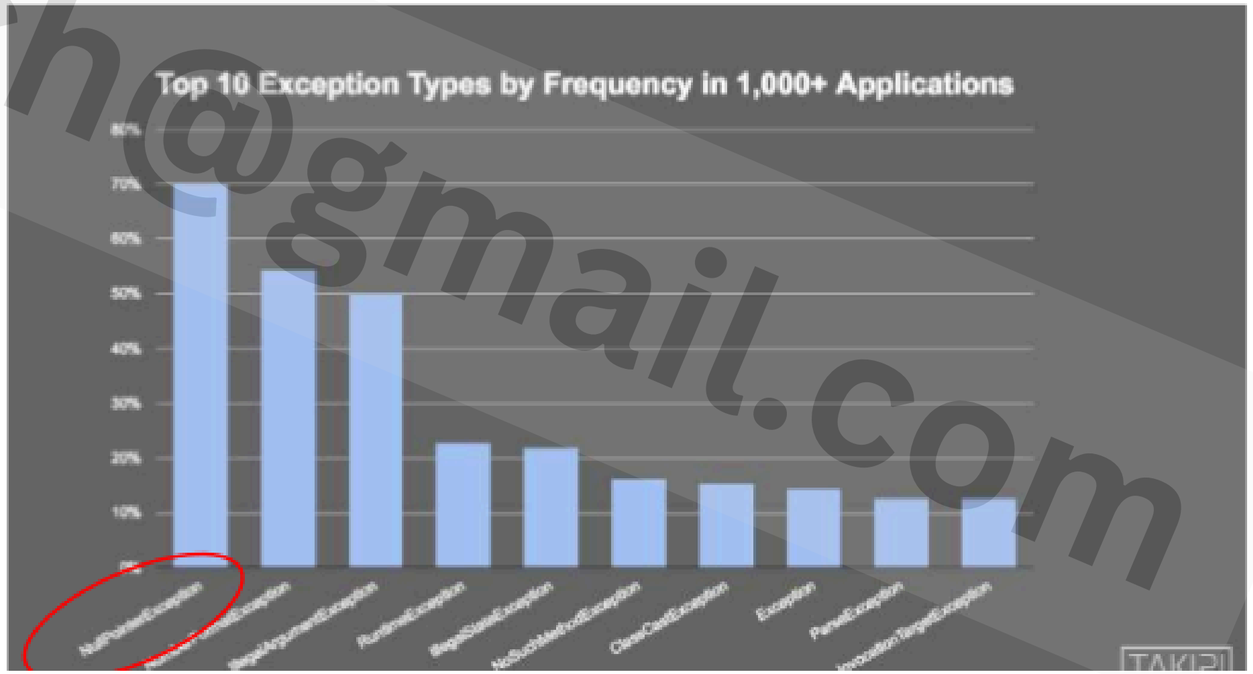
Need of Optional

The “Billion-Dollar mistake”

- Tony Hoare the creator of the null reference
- Created while designing ALGOL W
- Meant to model the absence of a value
- “… simply because it was so easy to implement”
- The cause of the dreaded Null Pointer Exceptions

NullPointerException

- Thrown when we attempt to use null value
- The most thrown exception



Need of Optional

```
public class Person {
    private Car car;
    public Car getCar() { return car; }
}
public class Car {
    private Insurance insurance;
    public Insurance getInsurance() { return insurance; }
}
public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

```
public String getCarInsuranceName(Person person) {
        return person.getCar().getInsurance().getName();
}
```

Need of Optional

```
public String getCarInsuranceName(Person person) {
      if (person != null) {
            Car car = person.getCar();
            if (car != null) {
                  Insurance insurance = car.getInsurance();
                  if(insurance != null) {
                        return insurance.getName();
                  }
            }
      }
      return "Unknown";
}
```

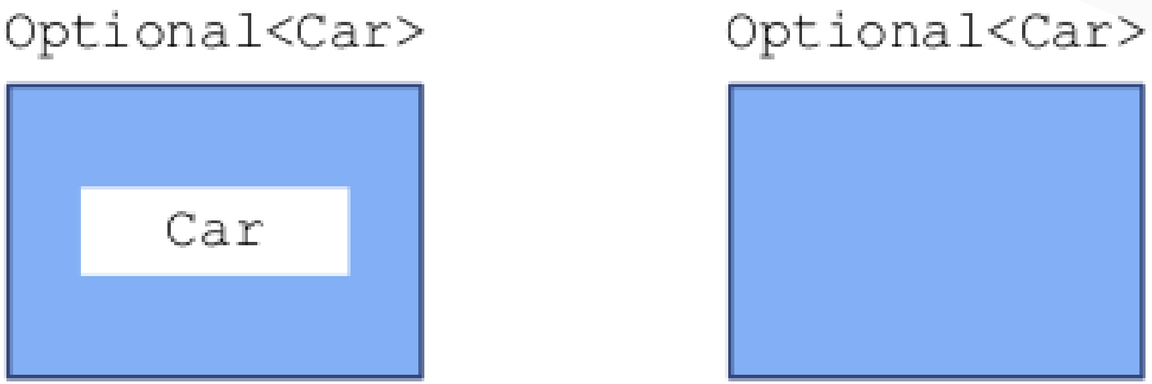
```
public String getCarInsuranceName(Person person) {
      if (person == null) {
            return "Unknown";
      }
      Car car = person.getCar();
      if (car == null) {
            return "Unknown";
      }
      Insurance insurance = car.getInsurance();
      if(insurance == null) {
            return "Unknown";
      }
      return insurance.getName();
}
```

So What's wrong with null

- It’s a source of error
  - NullPointerException
- It bloats code
  - Worsens readability
- It meaningless
  - Has no semantic meaning
- It breaks Java’s philosophy
  - Java hides pointers from developers, except in one case

Optional

- New class located at `java.util.Optional<T>`
- Encapsulates optional values
- It wraps the value if it's there, and is empty if it isn't
  - `Optional.empty()`
- Signals that a missing value is acceptable
- We can't deference `null`, but we can dereference `Optional.empty()`



Progressive Refactoring

```
public class Person {
    private Optional<Car> car;
    public Optional<Car> getCar() { return car; }
}
public class Car {
    private Optional<Insurance> insurance;
    public Optional<Insurance> getInsurance() { return insurance; }
}
public class Insurance {
    private String name;
    public String getName() { return name; }
}
```


rgupta.mtech@gmail.com

Progressive Refactoring

How do we use Optional?

```
Optional<Car> optCar = Optional.ofNullable(car);
Car realCar = optCar.get(); DO NOT DO THIS! (ANTI-PATTERN)
```

- Throws a `NoSuchElementException` if empty
- Same conundrum as using null

Creating Optional objects

- Empty Optional
  - `Optional<T> opt = Optional.empty();`
  - `Optional<Car> optCar = Optional.empty();`
- Optional from a non-null value
  - `Optional<T> opt = Optional.of(T value);`
  - `Optional<Car> optCar = Optional.of(car);`
- Optional from a null value
  - `Optional<T> opt = Optional.ofNullable(T value);`
  - `Optional<Car> optCar = Optional.ofNullable(car);`

Chaining Optionals

Get the name of an insurance policy that a person may have.

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

```
Optional<Person> optPerson = Optional.ofNullable(person)
Optional<String> name = optPerson.map(Person::getCar)
                                 .map(Car::getInsurance)
                                 .map(Insurance::getName);
```

- Doesn't compile
- `getCar` returns `Optional<Car>`
- Results in `Optional<Optional<Car>>`

Chaining Optionals

Get the name of an insurance policy that a person may have.

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

```
Public String getCarInsuranceName(Optional<Person> person) {
    return person.flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 // default value if Optional is empty
                 .orElse("Unknown");
}
```


rgupta.mtech@gmail.com

Chaining Optionals

Sometimes you need to throw an exception if a value is absent.

```
Public String getCarInsuranceName(Optional<Person> person, int minAge) {
    return person.filter(age -> p.getAge() >= minAge)
                 .flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 // Throw an appropriate exception
                 .orElseThrow(IllegalStateException::new);
}
```

Methods Optional class

Method	Description
empty()	Returns an empty **Optional** instance.
equals(Object obj)	Indicates whether some other objects is “equal to” this Optional.
filter(Predicate<? super T>predicate)	If a value is present, and this value matches the given predicate, return an **Optional** describing the value, otherwise return an empty **Optional**.
flatMap(Function<? super T, Optional<U> mapper)	If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional.
get()	If a value is present in this Optional, returns the value, otherwise throws **NoSuchElementException**.
hashCode()	Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.
ifPresent(Consumer<? Super T>consumer)	If a value is present, invoke the specified consumer with the value, otherwise do nothing.
isPresent()	Returns true if there is a value present, otherwise false.

Case study Stream processing

Data setup

```
public class Book {
    private List<Author> authors;
    private String title;
    private int pages;
    private Genre genre;
    private int year;
    private String Isbn;
}
```

```
public class Author {
    private String name;
    private String lastName;
    private String country;
}

public enum Genre {
    NOVEL, SHORT_NOVEL, NON_FICTION;
}
```

Functional interfaces Quick recap

▸ **We don't need to write all the functional interfaces because Java 8 API defines the basic ones in java.util.function package**

Functional interface	Descriptor	Method name
Predicate<T>	T → boolean	test()
BiPredicate<T, U>	(T, U) → boolean	test()
Consumer<T>	T → void	accept()
BiConsumer<T, U>	(T, U) → void	accept()
Supplier<T>	() → T	get()
Function<T, R>	T → R	apply()
BiFunction<T, U, R>	(T, U) → R	apply()
UnaryOperator<T>	T → T	identity()
BinaryOperator<T>	(T, T) → T	apply()

▸ **So we did not need to write the BookFilter interface, because the Predicate interface has exactly the same descriptor**

Stream Operations Quick recap

Operation	Operation type	Return type
filter(Predicate<T>)	intermediate	Stream<T>
map(Function <T, R>)	intermediate	Stream<R>
flatMap(Function <T, R>)	intermediate	Stream<R>
distinct()	intermediate	Stream<T>
sorted(Comparator<T>)	intermediate	Stream<T>
peek(Consumer<T>)	intermediate	Stream<T>
limit(int n)	intermediate	Stream<T>
skip(int n)	intermediate	Stream<T>
reduce(BinaryOperator<T>)	terminal	Optional<T>
collect(Collector<T, A, R>)	terminal	R
forEach(Consumer<T>)	terminal	void
min(Comparator<T>)	terminal	Optional<T>
max(Comparator<T>)	terminal	Optional<T>
count()	terminal	long
anyMatch(Predicate<T>)	terminal	boolean
allMatch(Predicate<T>)	terminal	boolean
noneMatch(Predicate<T>)	terminal	boolean
findFirst()	terminal	Optional<T>
findAny()	terminal	Optional<T>

Streams Problems

We need all the books with more than 400 pages

```
List<Book> longBooks =
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

We need the top three longest books

```
List<Book> top3LongestBooks =
books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).limit(3).Collect( toList());
```

Streams Problems

We need from the fourth to the last longest books.

```
List<Book> fromFourthLongestBooks =
books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).skip(3).collect(toList());
```

We need to get all the publishing years

```
List<Integer> publishingYears =
  books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

Streams Problems

We need all the authors

```
Set<Author> authors =
    books.stream().flatMap(b -> b.getAuthors().stream()).distinct().collect(toSet());
```

We need all the origin countries of the authors

```
Set<String> countries =
    books.stream().flatMap(b -> b.getAuthors().stream())
    .map(author -> author.getCountry()).distinct().collect(toSet());
```

Streams Problems

We want the most recent published book.(Hint Optional)

```
Optional<Book> lastPublishedBook =
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

We want to know if all the books are written by more than one author

```
boolean onlyShortBooks =
        books.stream().allMatch(b -> b.getAuthors().size() > 1);
```

Streams Problems

We want one of the books written by more than one author.

```
Optional<Book> multiAuthorBook =
        books.stream().filter((b -> b.getAuthors().size() > 1)).findAny();
```

We want the total number of pages published(Hint reduction).

```
Integer totalPages =
        books.stream().map(Book::getPages).reduce(0, (b1, b2) -> b1 + b2);
```

```
Optional<Integer> totalPages =
        books.stream().map(Book::getPages).reduce(Integer::sum);
```

Streams Problems

We want to know how many pages the longest book has.

```
Optional<Integer> longestBook =
      books.stream().map(Book::getPages).reduce(Integer::max);
```


rgupta.mtech@gmail.com

The Collector interface

The Collector interface was introduce to give developers a set of methods for reduction operations

Method	Return type
toList()	List<T>
toSet()	Set<t>
toCollection()	Collection<T>
counting()	Long
summingInt()	Long
averagingInt()	Double
joining()	String
maxBy()	Optional<T>
minBy()	Optional<T>
reducing()	...
groupingBy()	Map<K, List<T>>
partioningBy()	Map<Boolean, List<T>>

rgupta.mtech@gmail.com

Collector Streams Problems

We want the average number of pages of the books.

```
Double averagePages =
    books.stream().collect(averagingInt(Book::getPages));
```

We want all the titles of the books

```
String allTitles =
    books.stream().map(Book::getTitle).collect(joining(", "));
```

The Collector interface

We want the book with the higher number of authors?

```
Optional<Book> higherNumberOfAuthorsBook =
        books.stream().collect(maxBy(comparing(b -> b.getAuthors().size())));
```

Stream grouping Problems

We want a Map of book per year

```
Map<Integer, List<Book>> booksPerYear =
        Setup.books.stream().collect(groupingBy(Book::getYear));
```

We want a Map of how many books are published per year per genre.

```
Map<Integer, Map<Genre, List<Book>>> booksPerYearPerGenre =
    Setup.books.stream().collect(groupingBy(Book::getYear, groupingBy(Book::getGenre)));
```

Stream grouping Problems

We want to count how many books are published per year.

```
Map<Integer, Long> bookCountPerYear =
    Setup.books.stream().collect(groupingBy(Book::getYear, counting()));
```

Stream partitioning Problems

We want to classify book by hardcover

```
Map<Boolean, List<Book>> hardCoverBooks =
          books.stream().collect(partitioningBy(Book::hasHardCover));
```

We want to further classify book by Type.

```
Map<Boolean, Map<Genre, List<Book>>> hardCoverBooksByGenre =
    books.stream().collect(partitioningBy(Book::hasHardCover,groupingBy(Book::getGenre)));
```

Stream partitioning Problems

We want to count books with/without hardcover

```
Map<Boolean, Long> count =
        books.stream().collect(partitioningBy(Book::hasHardCover, counting()));
```