Core Java Training

Rajeev Gupta
Java Trainer & Consultant (MTech CS)
rgupta.mtech@gmail.com

Rajeev Gupta

FreeLancer Corporate Java JEE/ Spring Trainer

freelance • Institution of Electronics and Telecommunication Engineers IETE

New Delhi Area, India • 500+

---

1. Expert trainer for Java 8, GOF Design patterns, OOAD, JEE 7,Spring 5, Hibernate 5 ,Spring boot, microservice, netflix oss, Spring cloud, angularjs, Spring MVC, Spring Data, Spring Security, EJB 3, JPA 2, JMS 2, Struts 1/2, Web service

2. Helping technology organizations by training their fresh and senior engineers in key technologies and processes.

3. Taught graduate and post graduate academic courses to students of professional degrees.

I am open for advance java training /consultancy/ content development/ guest lectures/online training for corporate / institutions on freelance basis

Day-1: Introdcution to Java, JVM, procedural programming, Arrays

- Introduction to OO Concepts
- Introduction to UML
- Pillars of OO: Inheritance - Polymorphism - Abstraction - Encapsulation
- JVM basics: JDK,JRE and JVM
- Procedural programming: if else, looping, swith etc
- Array, defining usages 1D/2D arrays
- Lab Assignments

Day 2: Object Orientation & JVM introduction

- Creating Class, Object, constructor, init paramaters
- Static variable and method
- Concepts of packages, Access specifier
- Inheritance, Types of inheritance in Java, Inheriting Data Member and Methods
- Role of Constructors in inheritance,Overriding super Class methods, super
- Hands On & Lab

Day 3: Advanced Class Features

- Loose coupling and high cohesion
- composition, aggrigation, inheritance, basic of uml
  - Abstract classes and methods
  - Relationship between classes- IS-A, HAS-A, USE-A
  - Interface Vs Abstract, when to use what?
  - Final classes and methods
  - Interfaces, loose coupling and high cohesion
  - SOLID principles, Square – rectangle problem
  - Hands On & Lab

Day-4:Strings,Wrapper classes, Immutability, Inner classes,Lambda expression, Java 5 features , IO, Exception Handling

- String class: Immutability and usages, stringbuilder and stringbuffer
- Wrapper classes and usages, Java 5 Language Features
- AutoBoxing and Unboxing, Enhanced For loop, Varargs, Static Import, Enums
- Inner classes: Regular inner class, method local inner class, annonymous inner class
- Char and byte oriented streams
- BufferedReader and BufferedWriter
- File handling
- Object Serialization and IO [ObjectInputStream / ObjectOutputStream]
- Exception Handling, Types of Exception, Exception Hierarchy, Exception wrapping and re throwing
- Hands On & Lab

Day 5: Introduction to Java threads, thread life cycle, synchronization, dead lock

- Program vs process
- Thread as LWP
- Creating and running thread
- Thread life cycle
- Need of synchronization
- Producer consumer problem
- dead lock
- Hands on & Lab

Day 6:Java Collection, Generics

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

Day-7: Annotation, Java Reflection

- Introduction to java reflection
- Java Annotation, JDK annotation, creating custom annotation, Annotation and reflection
- Introduction to Design pattern
- Hands on & Lab

Day 8: GOF design patterns

- Pattern categories: Creational, Structural, Behavioral
- Creational Patterns: Singleton, Factory, Builder, Prototype
- Structural Patterns: Decorator, Facade Pattern, Proxy
- Behavioral Patterns: Iterator, Observer, Strategy,Template Method
- Hands on & Lab

rgupta.mtech@gmail.com

Day-1: Introdcution to Java, JVM, procedural programming, Arrays

- Introduction to OO Concepts
- Introduction to UML
- Pillars of OO: Inheritance - Polymorphism - Abstraction - Encapsulation
- JVM basics: JDK,JRE and JVM
- Procedural programming: if else, looping, swith etc
- Array, defining usages 1D/2D arrays
- Lab Assignments

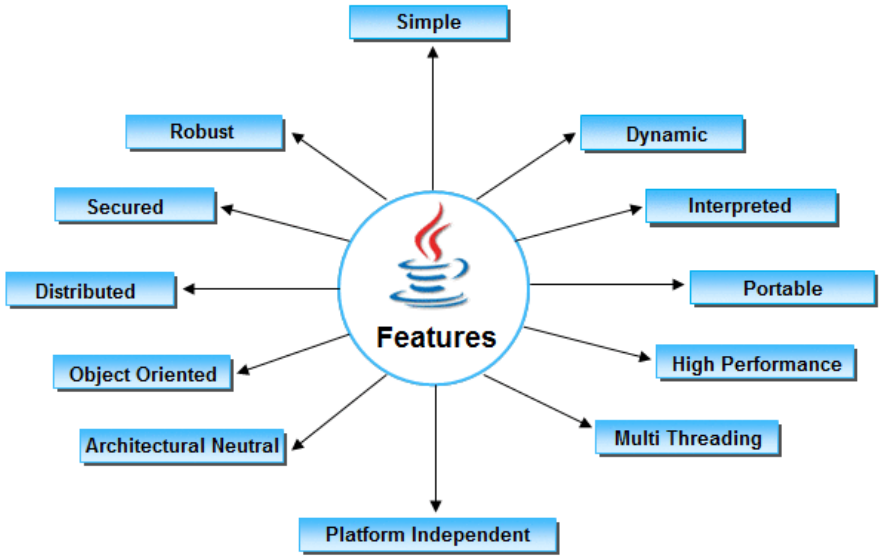
What is Java?

Java=OOPL+JVM+lib

- How Java is different then C++?

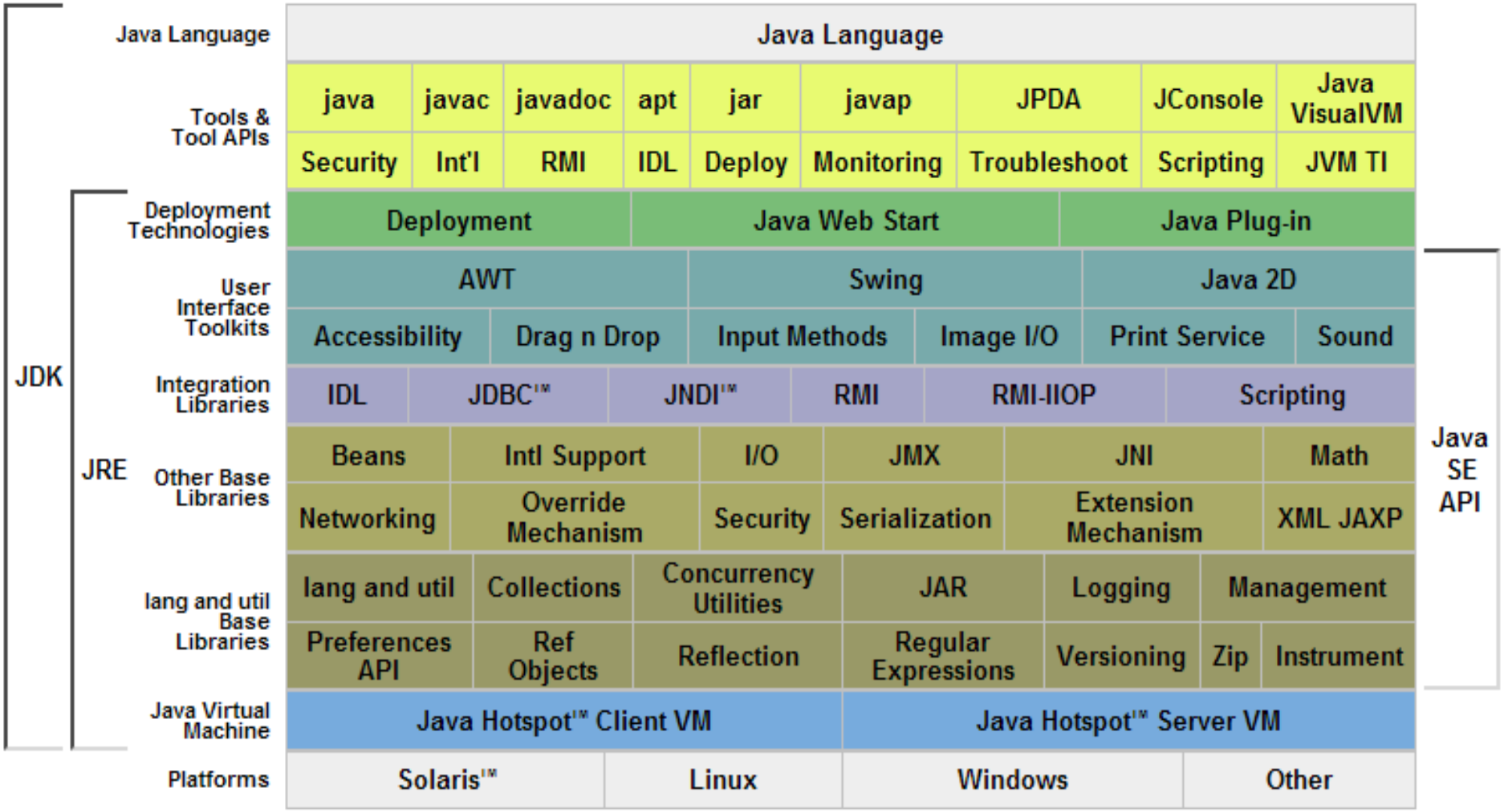
rgupta.mtech@gmail.com

How Sun define Java?



rgupta.mtech@gmail.com

Java Platform



rgupta.mtech@gmail.com

Versions

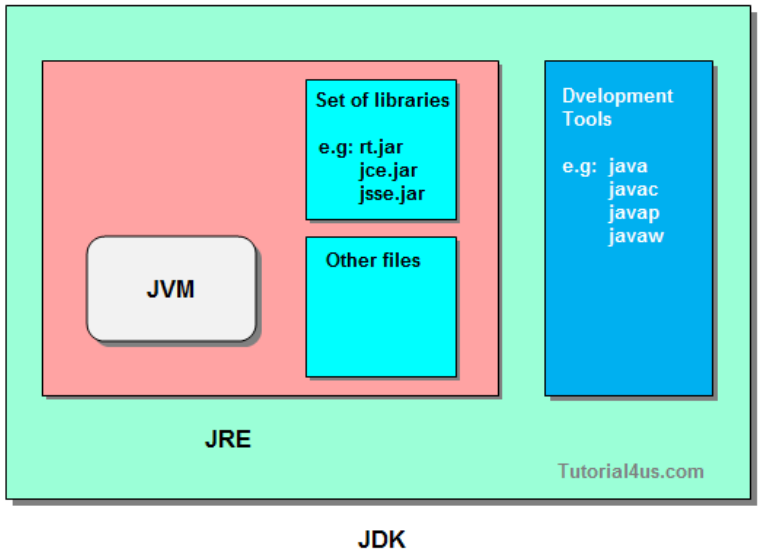
- JDK 1.0 (1995)
- JDK 1.1 (1997)
- J2SE 1.2 (1998)) Playground
- J2SE 1.3 (2000) Kestrel
- J2SE 1.4 (2002) Merlin
- J2SE 5.0 (2004) Tiger
- Java SE 6 (2006) Mustang
- Java SE 7 (2011) Dolphin
- Java SE 8 (2014)

Called Java2

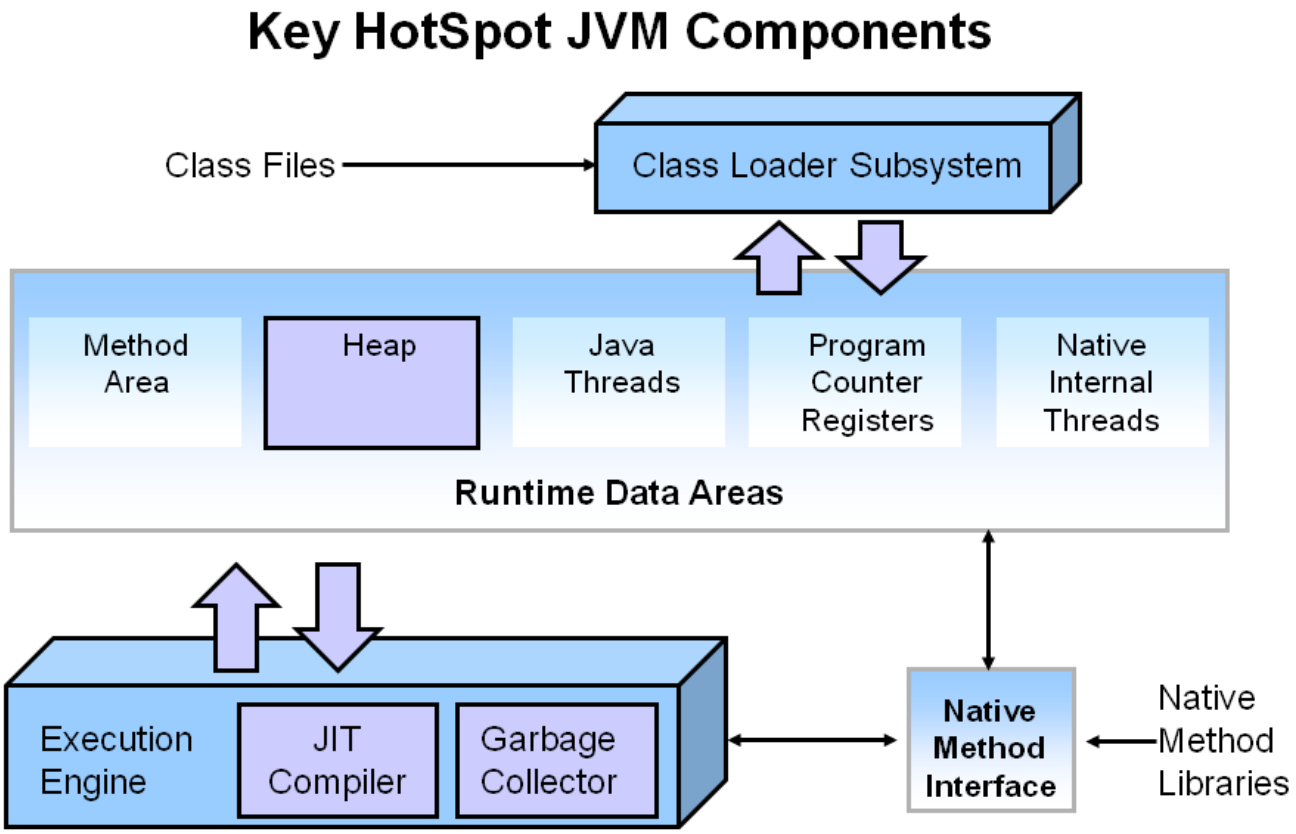
Version for the session

Some common buzzwords

- JDK
  - Java development kit, developer need it
- Byte Code
  - Highly optimize instruction set, designed to run on JVM
  - Platform independent
- JRE
  - Java Runtime environment, Create an instance JVM, must be there on deployment machine.
  - Platform dependent
- JVM
  - Java Virtual Machine JVM
  - Pick byte code and execute on client machine
  - Platform dependent…



Understanding JVM

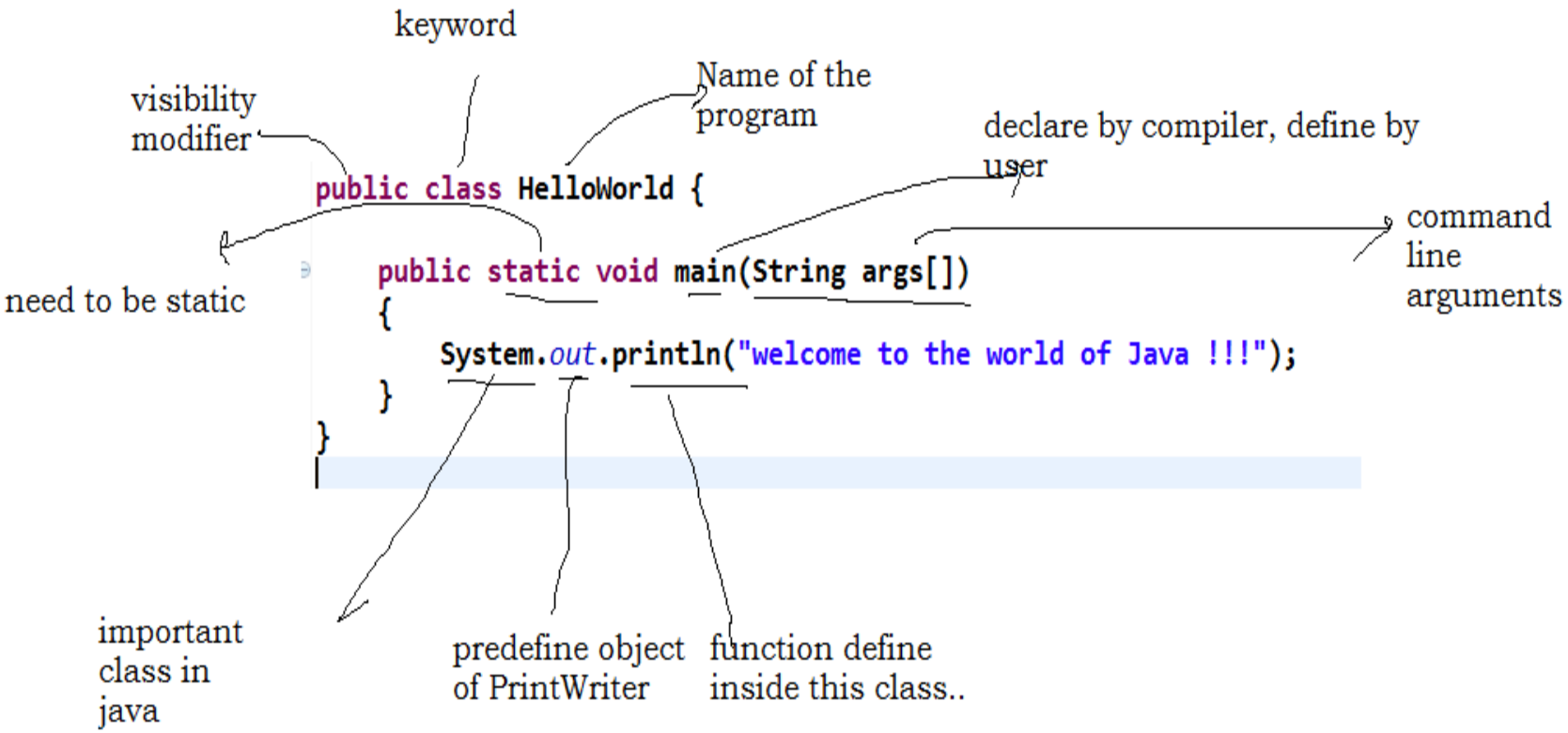


Hello World

- Lab set-up
  - Java 1.8
  - Eclipse Luna

```
public class HelloWorld {

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

Analysis of Hello World !!!

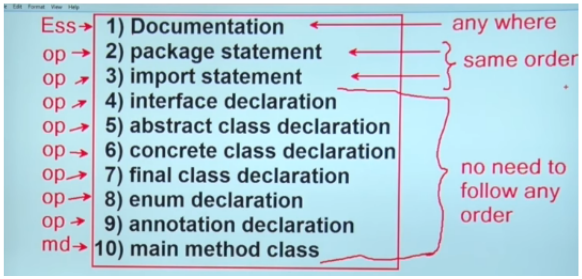


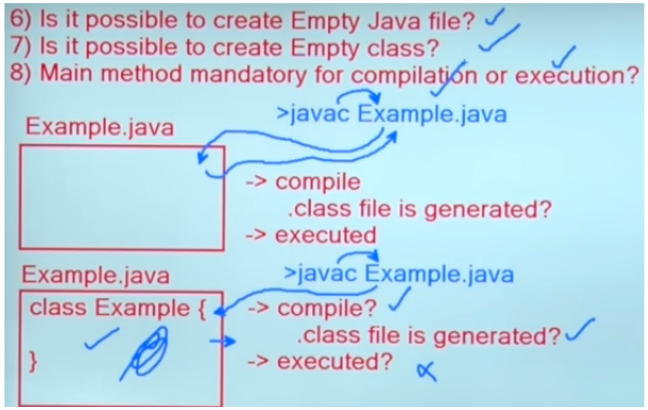
Some basics rules about java classes

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?

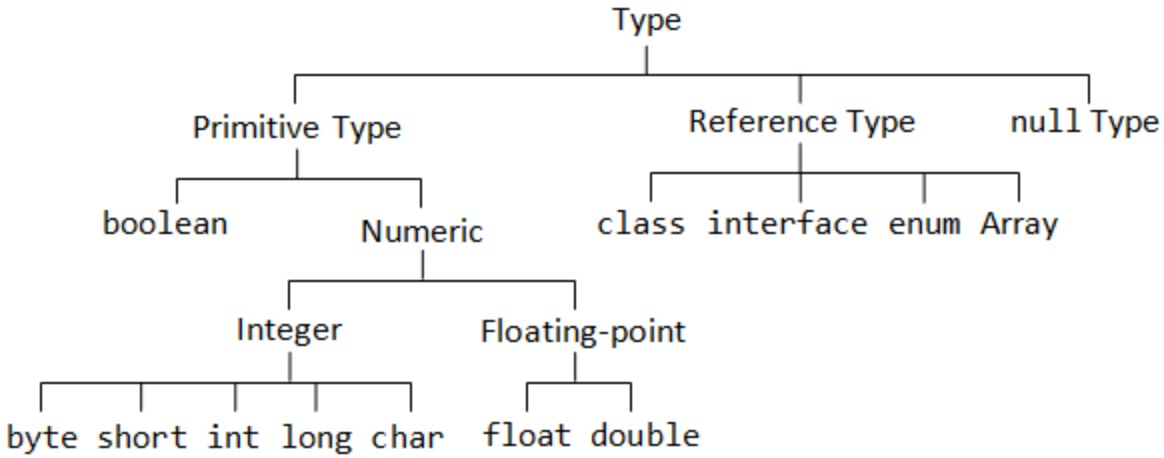




rgupta.mtech@gmail.com

Java Data Type

- 2 type
  - Primitive data type
  - Reference data type



Primitive data type

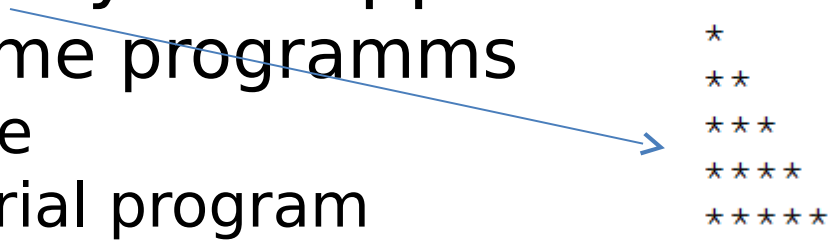
- boolean — either true of false
- char — 16 bit Unicode 1.1
- byte — 8-bit integer (signed)
- short — 16-bit integer (signed)
- int — 32-bit integer (signed)
- long — 64-bit integer (singed)
- float — 32-bit floating point (IEEE 754-1985)
- double — 64-bit floating point (IEEE 754-1985)

Java uses Unicode to represent characters internally

Procedural Programming

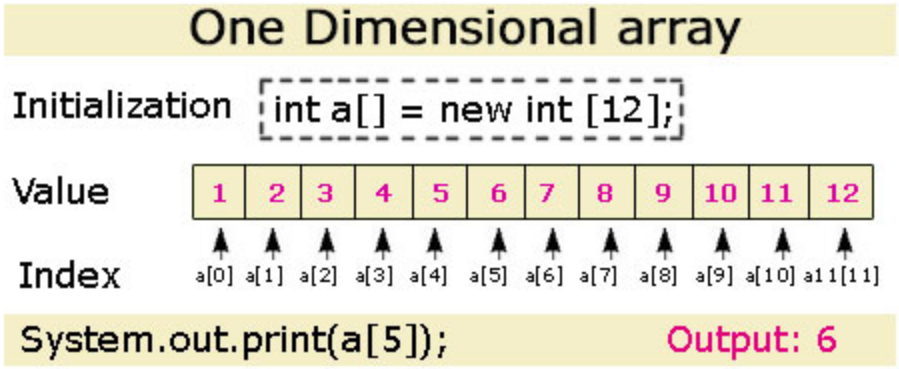
- if..else
- switch
- Looping; for, while, do..while as usual in Java as in C/C++

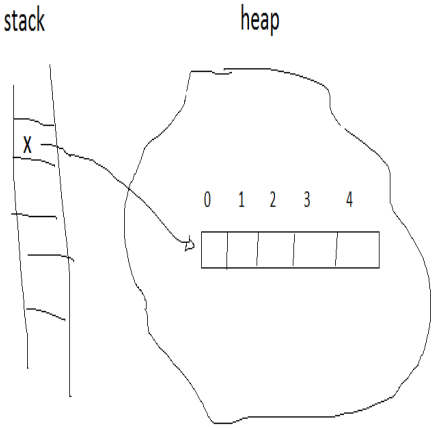
- Don’t mug the program/logic
- Follow dry run apprc
  - Try some programms
    - Create
    - Factorial program
    - Prime No check
    - Date calculation



Array

- How Java array different from C/C++ array?





Java Array: its different then C/C++

Object technologies

- 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 an Object?

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

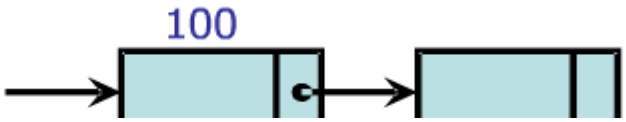
Tom's Car

  - Conceptual entity

US$100,000,000,000

Bill Gate's bank account

  - Software entity



rgupta.mtech@gmail.com

Class

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

Object factory

Cookie Cutter

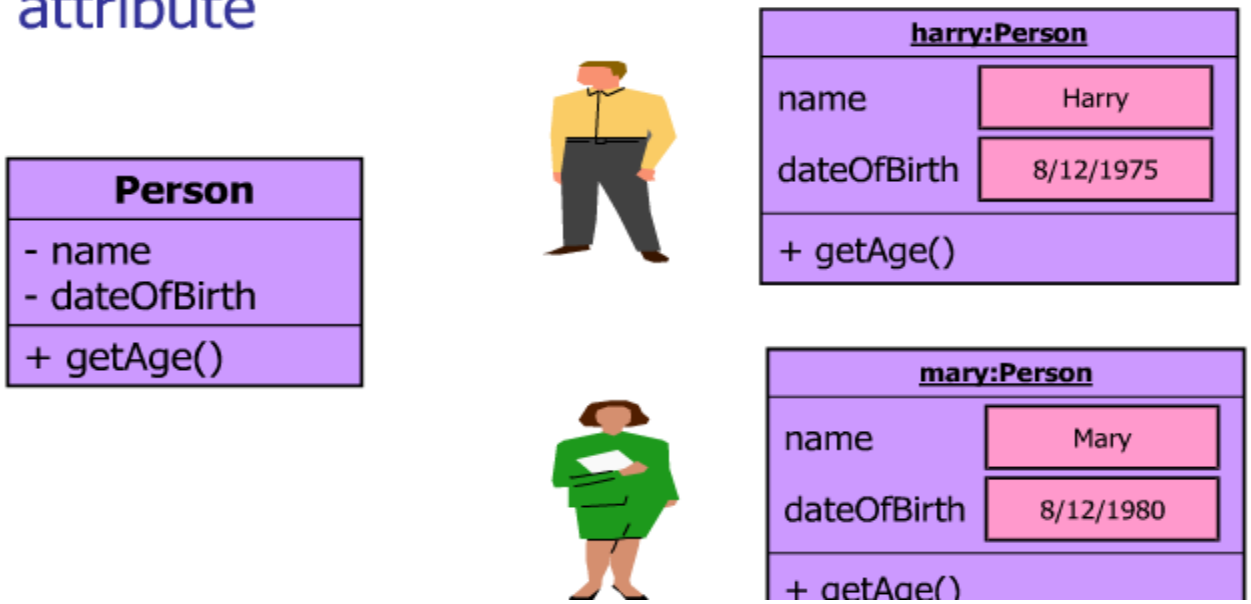
```
public class StudentTest {
  public static void main(String[] args) {
    Student s1 = new Student();
    Student s2 = new Student();
  }
}
```

```
- class -

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

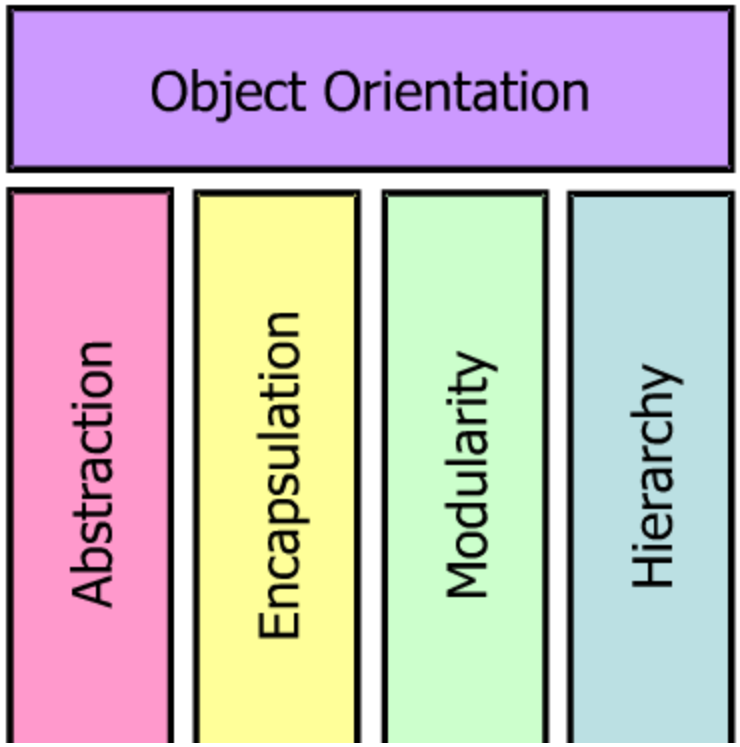
rgupta.mtech@gmail.com

Classes and Object

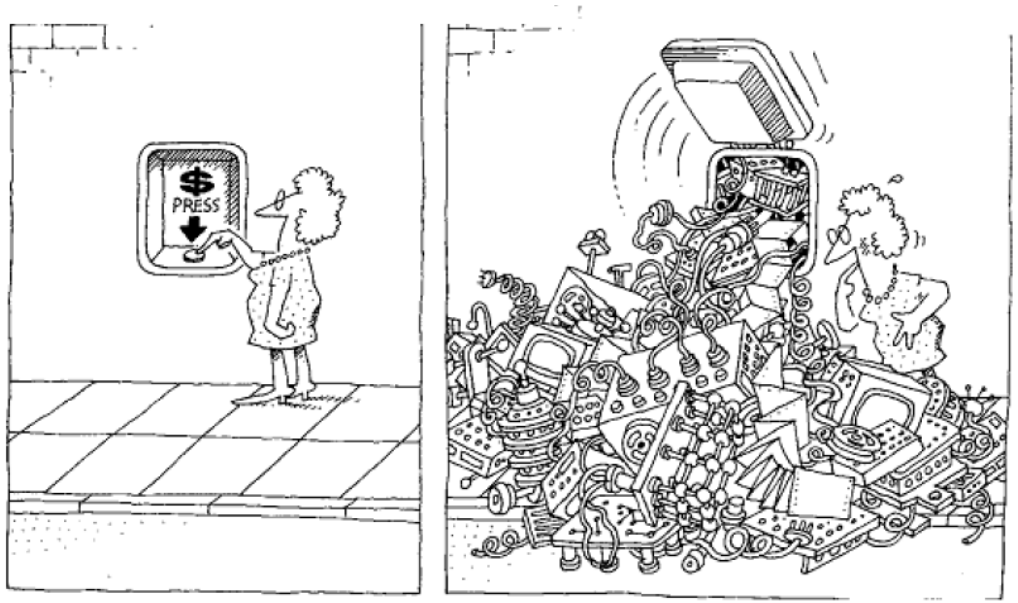
- All the objects share the same attribute names and methods with other objects of the same class
- Each object has its own value for each of the attribute



Basic principles of OO



Dealing with Software complicity:Abstraction



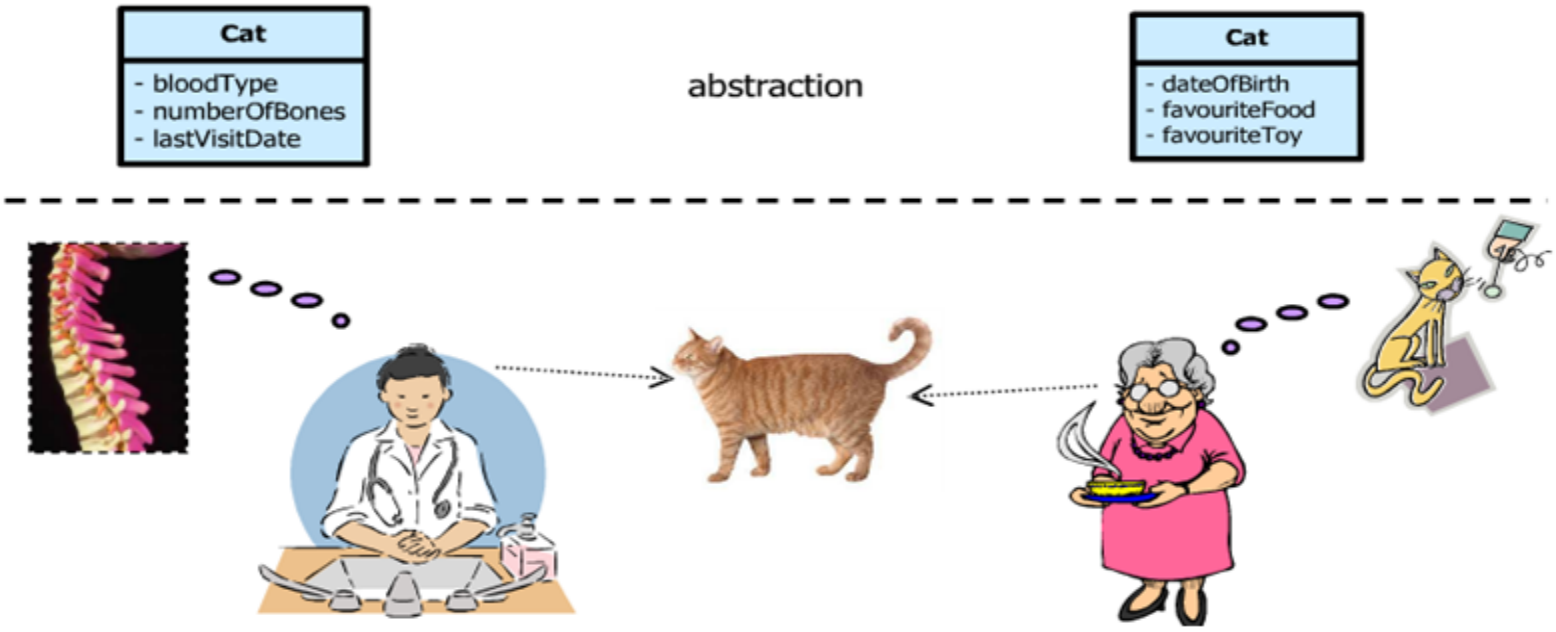
The task of the software development team is to engineer the illusion of simplicity.

Abstraction

- Fundamental ways that we use to cope with complexity
- "abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences" -Hoare
- An abstraction denotes the **essential characteristics of an object that distinguish it from all other kinds of objects** and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

Abstraction

- Determine the relevant properties and features while ignoring non-essential details

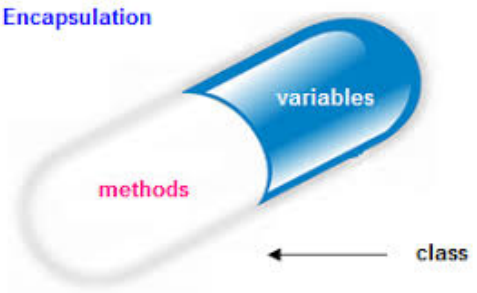


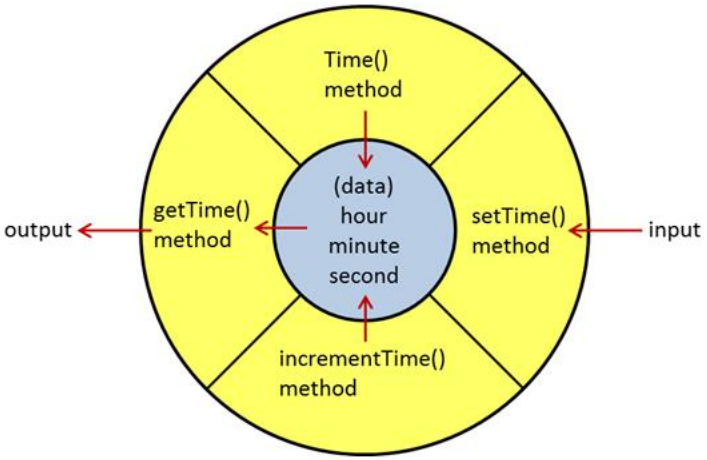
Encapsulation

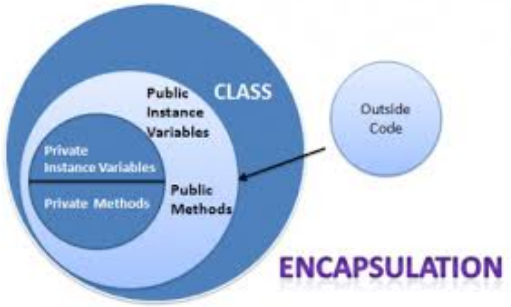
- Abstraction and encapsulation are *complementary concepts*: abstraction focuses upon the observable behavior of an object, whereas encapsulation focuses upon the implementation that gives rise to this behavior.
- Encapsulation is most often achieved through information hiding which is the process of hiding all the secrets of an object that do not contribute to its essentialcharacteristics; typically, the structure of an object is hidden, as well as the ,implementation of its methods.
- Information hiding is tool to achive encapsulation

Understanding encapsulation

```
class Hacker{
Account a= new Account ();
a.account_balance= -100;

}
```

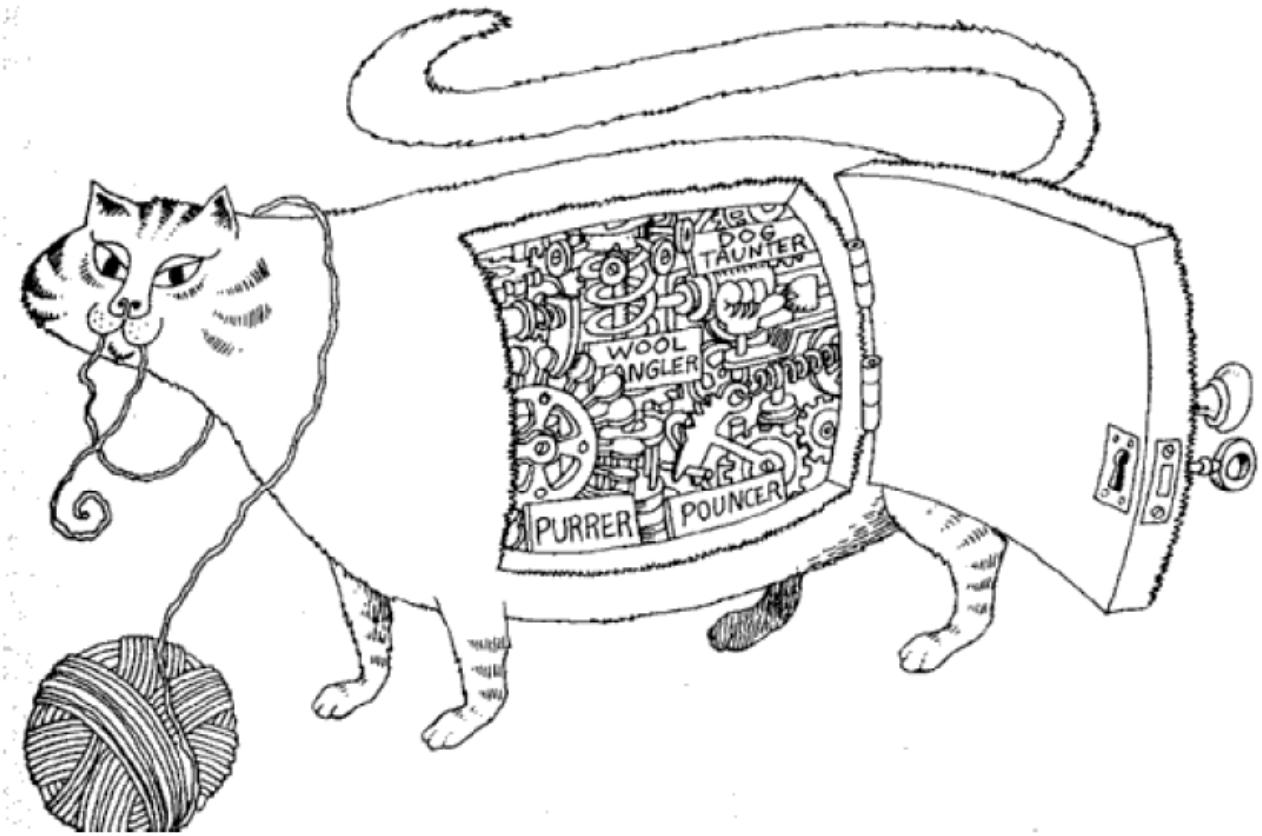






Encapsulation

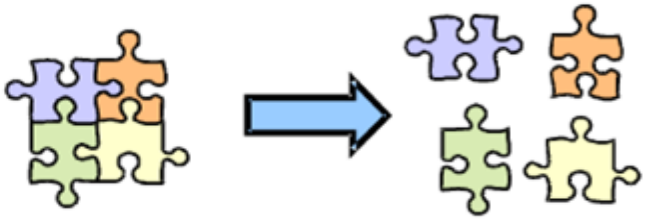
- Changing data in organized way, by using data hiding and appying business constraints
- Encapsulation= data hiding + constraints

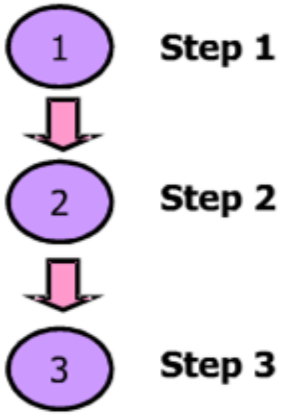


Encapsulation hides the details of the implementation of an object.

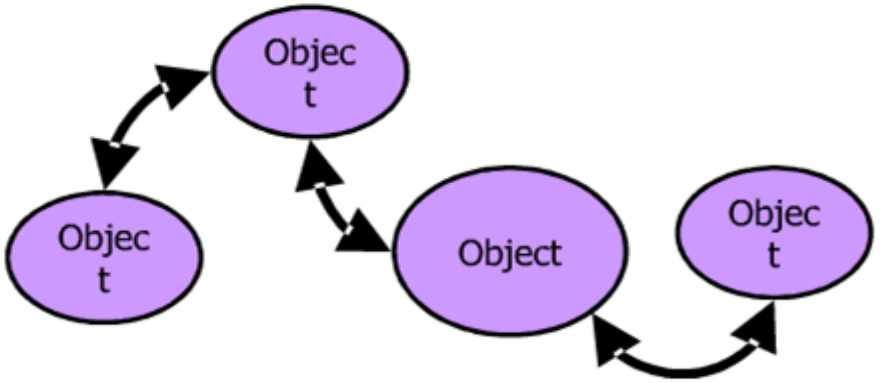
Modularity

- Break something complex into manageable pieces
  - Functional Decomposition
  - Object Decomposition





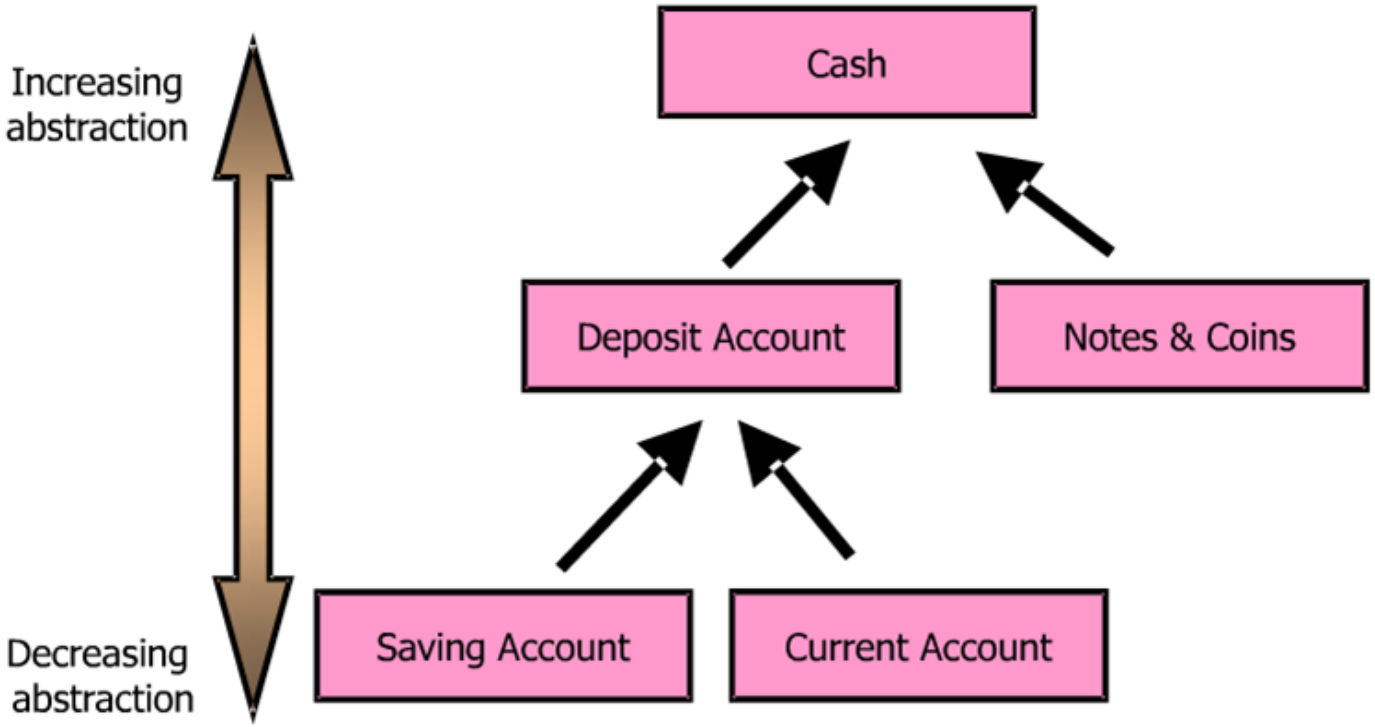
Functional Decomposition



Object Decomposition

Hierarchy

- Ranking or ordering of objects



rgupta.mtech@gmail.com

Day 2: Object Orientation & JVM introduction

- Creating Class, Object, constructor, init paramaters
- Static variable and method
- Concepts of packages, Access specifier
- Inheritance, Types of inheritance in Java, Inheriting Data Member and Methods
- Role of Constructors in inheritance,Overriding super Class methods, super
- Hands On & Lab

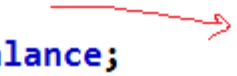
What can goes inside an class?

```
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
    }
}
```

Constructors: default, parameterized and copy

- 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
  - Refer next slide…

“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?

- Calling one constructor from another ?

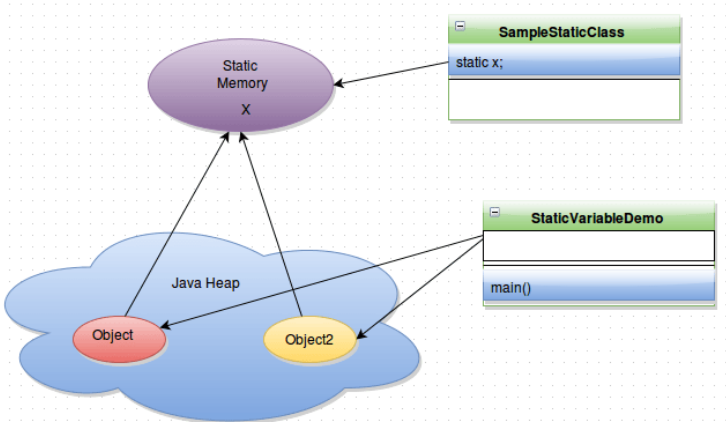
```
class Account{

    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) {
            //
```

Static method/variable

- 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?



Using static data..

```
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());
```

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;

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

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

}
```

code repetition.

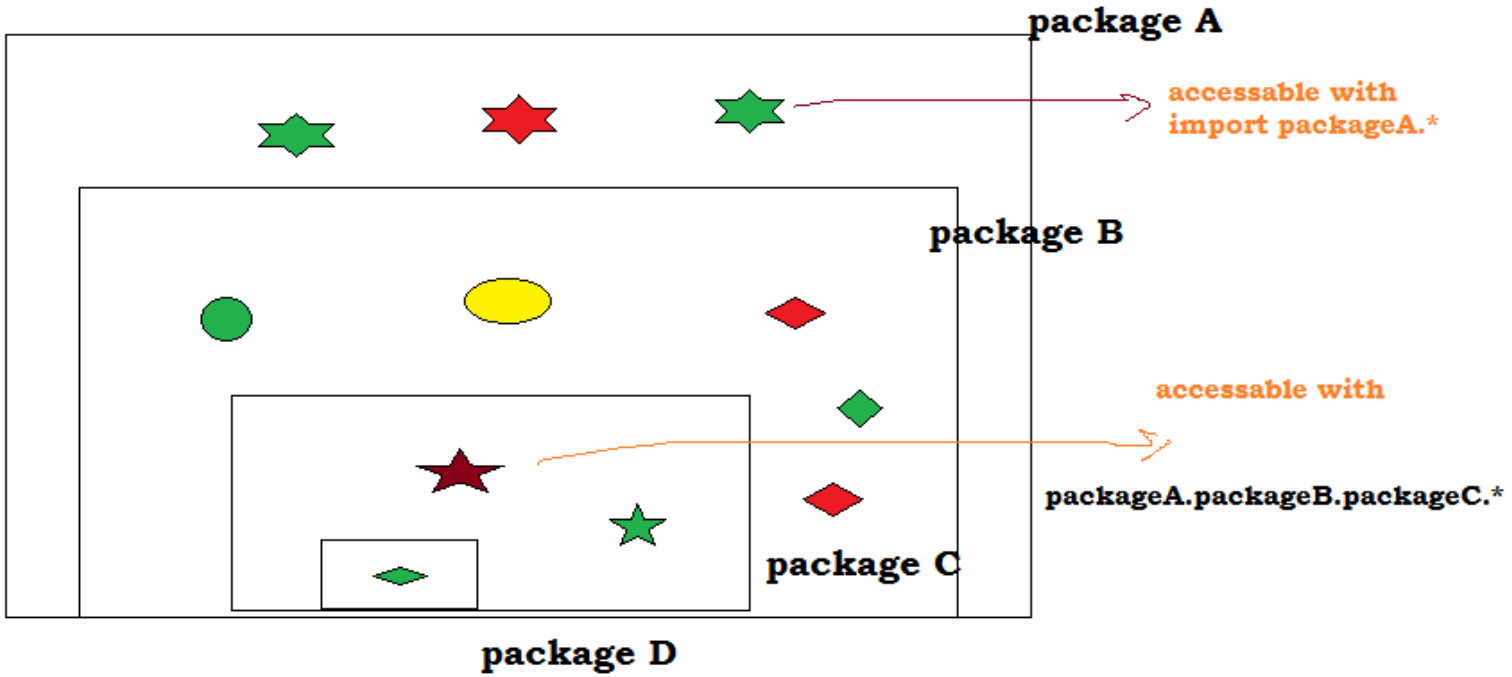
rgupta.mtech@gmail.com

Initialization block

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

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.

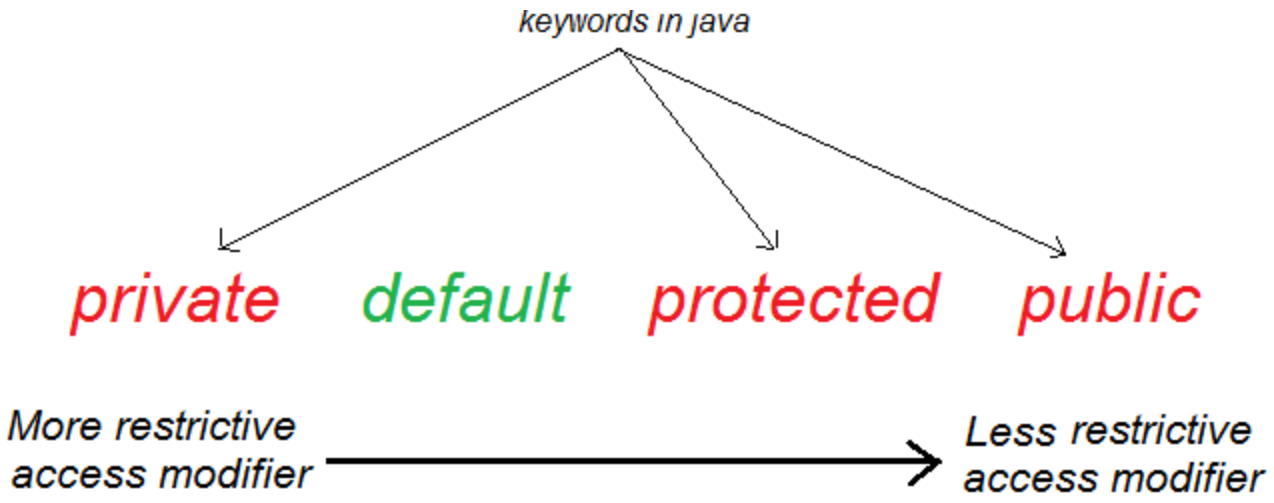


Java Foundation Packages

- Java provides a large number of classes groped into different packages based on their functionality.
- The six foundation Java packages are:
  - java.lang
    - Contains classes for primitive types, strings, math functions, threads, and exception
  - java.util
    - Contains classes such as vectors, hash tables, date etc.
  - java.io
    - Stream classes for I/O
  - java.awt
    - Classes for implementing GUI – windows, buttons, menus etc.
  - java.net
    - Classes for networking
  - java.applet
    - Classes for creating and implementing applets

Visibility Modifiers

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



Visibility Modifiers

- class A has default visibility hence can access in the same package only.

```
pack packA;

class A{
      public void foo(){
            }
}
```

```
pack packB;
import packA.*;

class B{
      public void boo(){
      A a=new A();
      }
}
```

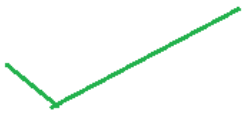
- Make class A public, then access it.

- Protected data can access in the same package and all the subclasses in other packages provide class itsef is public

```
pack packA;

public class A{
protected void foo(){
}
}
```

```
pack packB;
import packA.*;

class B{
      public void boo(){
      A a=new A();
      }
}
```

```
pack packB;
import packA.*;
class C extends A{

      public void foo2(){
            foo();
      }
}
```



Want to accept parameter from user?

java.util.Scanner (Java 1.5)

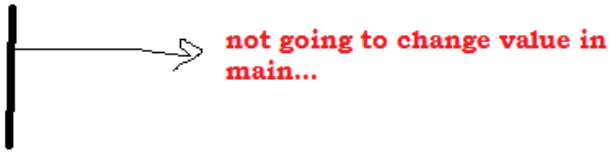
```
Scanner stdin = Scanner.create(System.in);

int n = stdin.nextInt();
String s = stdin.next();


boolean b = stdin.hasNextInt()
```

rgupta.mtech@gmail.com

Call by value

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

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

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



Call by reference

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

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

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

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

do not effect f1 , f2 in main

can change state of f1


DAY 3

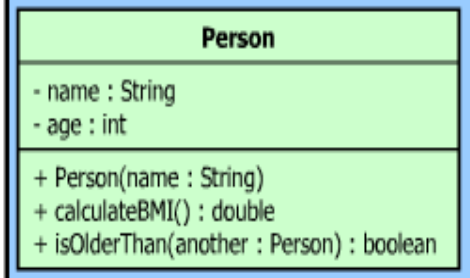
Day 3: Advanced Class Features

- Loose coupling and high cohesion
- composition, aggrigation, inheritance, basic of uml
  - Abstract classes and methods
  - Relationship between classes- IS-A, HAS-A, USE-A
  - Interface Vs Abstract, when to use what?
  - Final classes and methods
  - Interfaces, loose coupling and high cohesion
  - SOLID principles, Square – rectangle problem
  - Hands On & Lab

A bit about UML diagram...

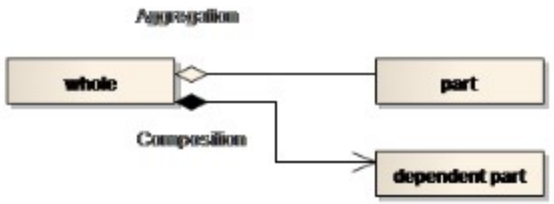
- UML 2.0 aka modeling language has 12 type of diagrams
- Most important once are class diagram, use case diagram and sequence diagram.
- You should know how to read it correctly
- This is not UML session... ☐

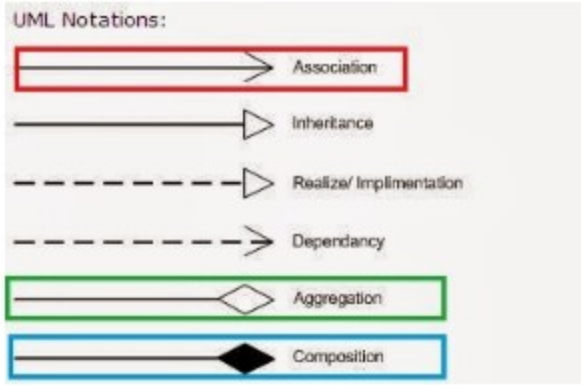
UML	Implementation

	`public class Person {`
	`  private String name;`
	`  private int age;`
	
	`  public Person(String name) { ...}`
	
	`  public double calculateBMI() { ... }`
	
	`  public boolean isOlderThan(Person another) {...}`
	`}`

Relationship between Objects

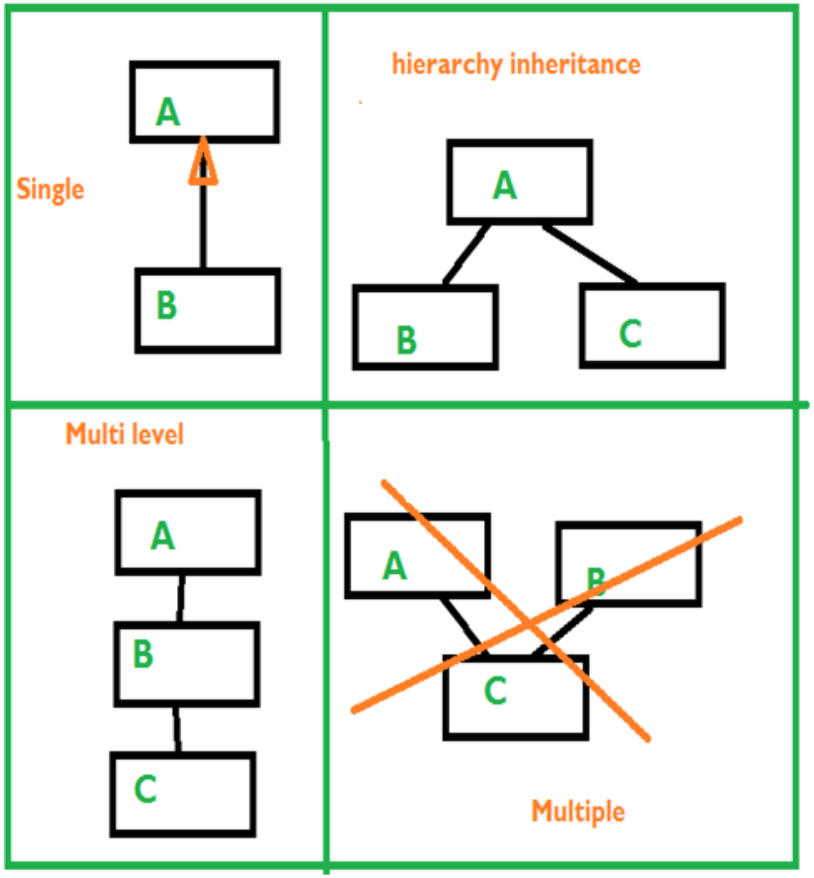
- USE-A
  - Passanger using metro to reach from office from home
- HAS-A (Association)
  - Compostion
    - Flat is part of Durga apartment
  - Aggregation
    - Ram is musician with RockStart musics group
- IS-A
  - Empoloyee is a person





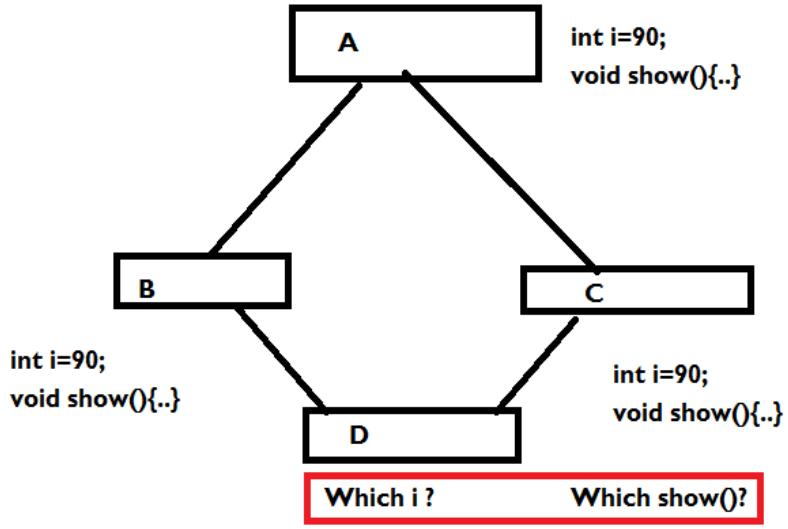
Inheritance

- ***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 sa
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

Be careful of overloading 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.**

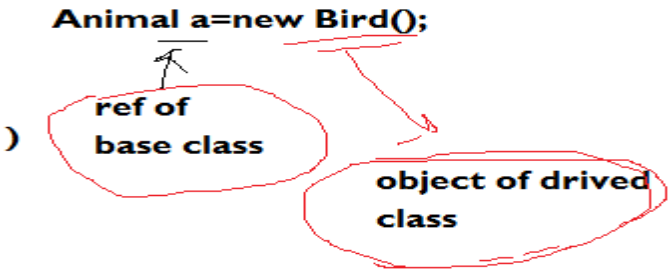
```
class BaseClass{
    public void getInvestAmount(int rate) {…}
}

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

Polymorphism

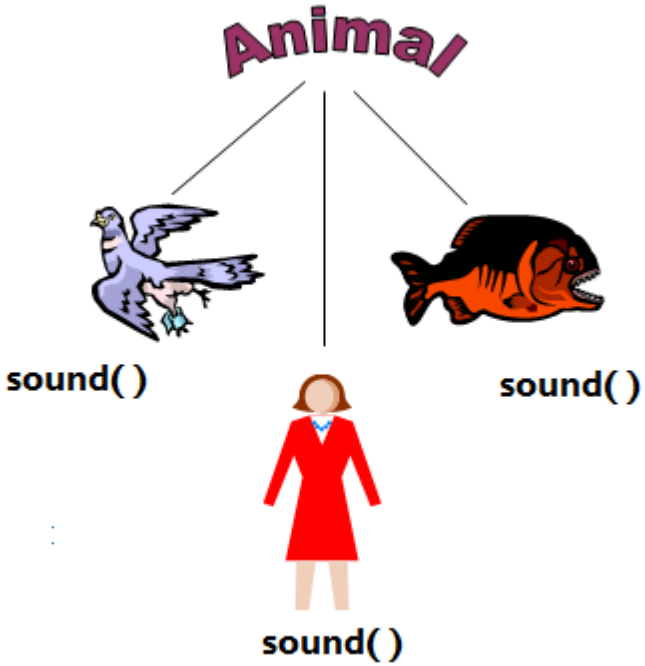
- Polymorphism=many forms of one things
- ***Substitutability***
- ***Overriding***
- ***Polymorphism means the ability of a single variable of a given type to be used to reference objects of*** different types, and automatically call the method that is specific to the type of object the variable references.

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...

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

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

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

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

```
abstract class Foo{
        }
```

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 mom

```
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

More example...

```
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
   }
}
```

Final

- 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….

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

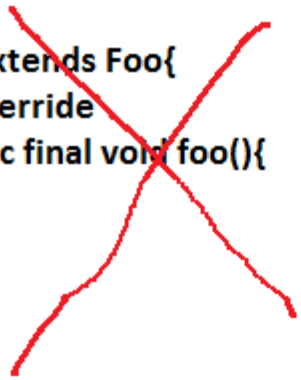
```
package cart;

public final class Beverage{

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

```
package examStuff;
import cart.*;

class Tea extends beverage{

}
```

Final Method

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

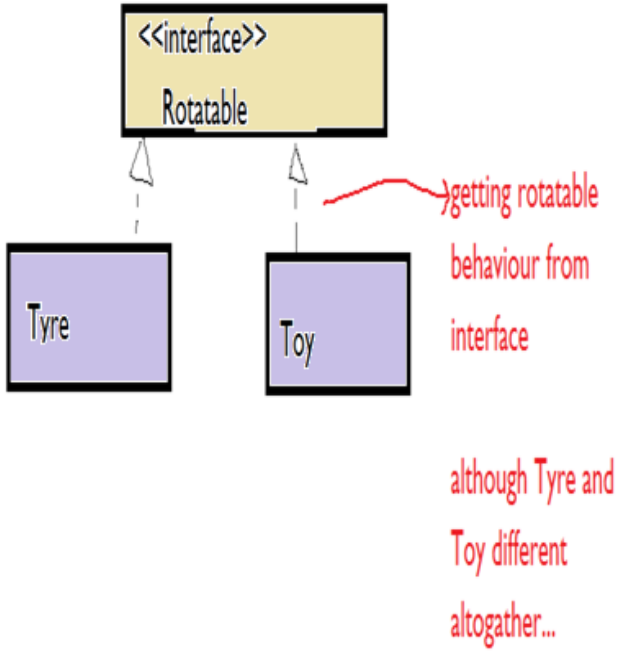
```
class Foo{

        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(){


        }
}
```



Interface?

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

Some Example ….



Interface?

- Rules
  - All interface methods are always public and abstract, whether we say it or not.
  - Variable declared inside interface are always public static and final
  - Interface method can’t be static or final
  - Interface cant have constructor
  - An interface can extends other interface
  - Can be used polymorphically
  - An class implementing an interface must implement all of its method otherwise it need to declare itself as an abstract class…

Implementing an interface...

What we declare

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

What compiler think...

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

All interface method must be implemented....

```
class Tyre implements Bouncable{

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

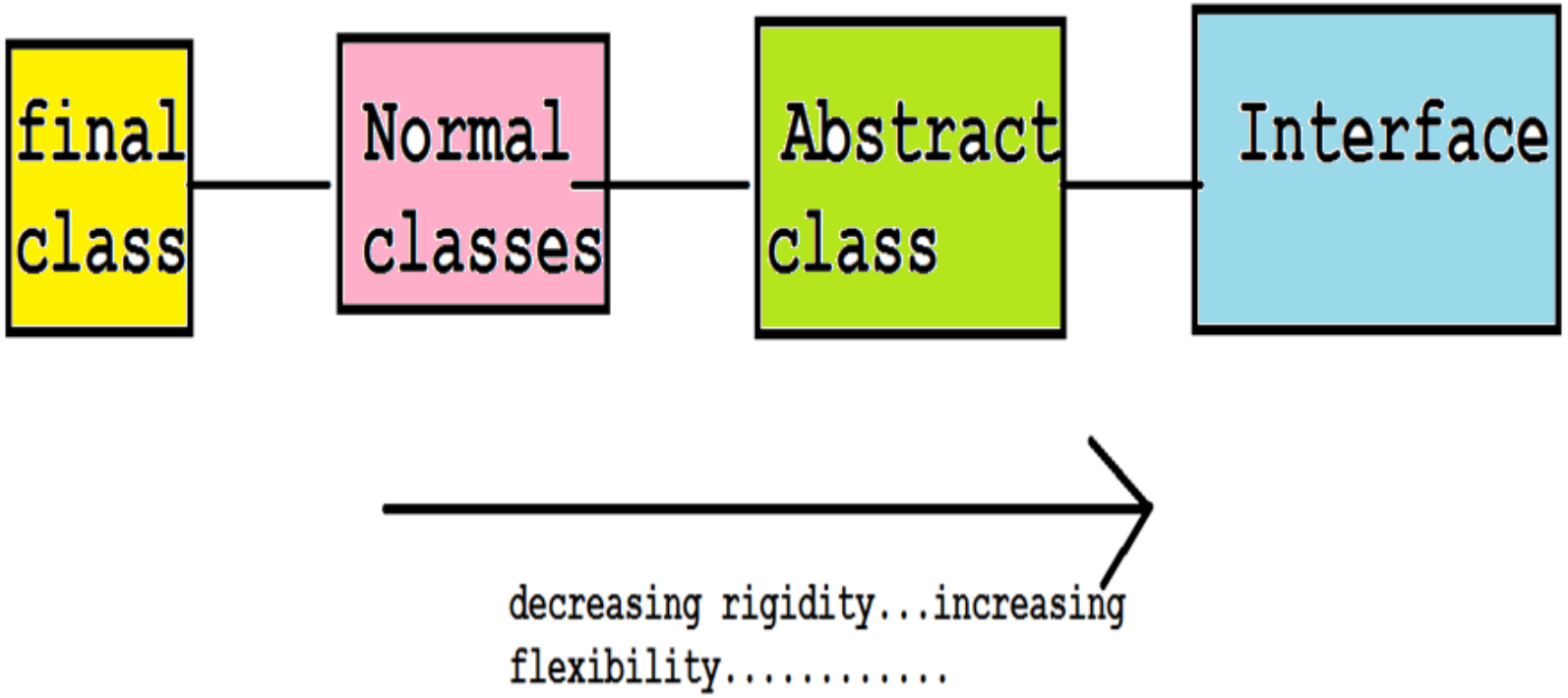
rgupta.mtech@gmail.com

Note

- 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();

Decreasing Rigidity..increasing Flexibility



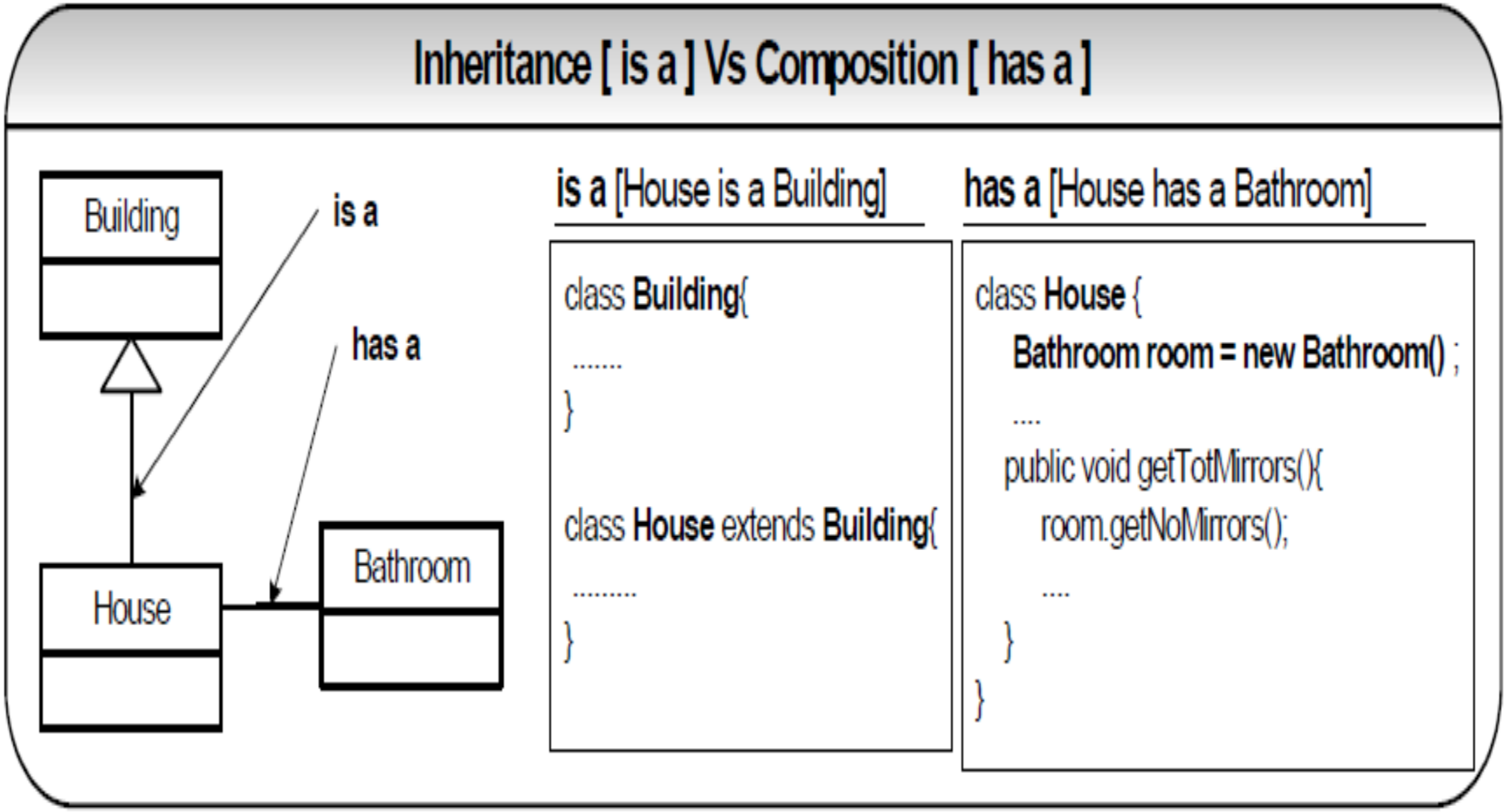
Type of relationship bw objects

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

ALWAYS GO FOR LOOSE COUPLING AND HIGH COHESION…

But HOW?

IS-A VS HAS-A



rgupta.mtech@gmail.com


DAY 4

Day-4:Strings,Wrapper classes, Immutability, Inner classes,Lambda expression, Java 5 features , IO, Exception Handling

- String class: Immutability and usages, stringbuilder and stringbuffer
- Wrapper classes and usages, Java 5 Language Features
- AutoBoxing and Unboxing, Enhanced For loop, Varargs, Static Import, Enums
- Inner classes: Regular inner class, method local inner class, annonymous inner class
- Char and byte oriented streams
- BufferedReader and BufferedWriter
- File handling
- Object Serialization and IO [ObjectInputStream / ObjectOutputStream]
- Exception Handling, Types of Exception, Exception Hierarchy, Exception wrapping and re throwing
- Hands On & Lab

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");

Immutability

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

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

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

Don't provide setter methods for variables

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

Initialize all the fields via a constructor performing deep copy.

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

How to make copy of an Object?

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

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

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

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

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

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

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

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

Various memory area of JVM

1. **Method area** ------ **Per JVM**
2. **heap area** ------ **Per JVM**
3. **stack area** ------ **Per thread**
4. **PC register area** ------ **Per thread**
5. **Native method area** ------ **Per thread**

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

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

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

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 pa
    argument
- Eight wrapper for eight primiti
  - Integer, Float, Double, Character

```
Integer it=new Integer(33);
int temp=it.intValue();
....
```

primitive==>object

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

object==>primitive

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

primitive ==>string

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

String==>Numeric object

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

rgupta.mtech@gmail.com

Boxing / Unboxing Java 1.5

- Boxing

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

- Unboxing

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

Java 5 Language Features (I)

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

rgupta.mtech@gmail.com

Enhanced For loop

- Provide easy looping construct to loop through array and collections

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

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

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

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

Varargs

- Java start supporting variable argument method in Java 1.5

```
class Foo{

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

- Discuss function overloading in case of Varargs and wrapper classes

Static Import

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

import static java.lang.Math.cos;

- Now rather then using
  - ***double r = Math.cos(Math.PI * theta);***
- We can use something like
  - ***double r = cos(PI * theta);*** – looks more readable ….

  - Avoid abusing static import like
    - *import static java.lang.Math.*;*

General guidelines to use static java import:

- 1) Use it to declare local copies of java constants
- 2) When you require frequent access to static members from one or two java classes

Enums

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

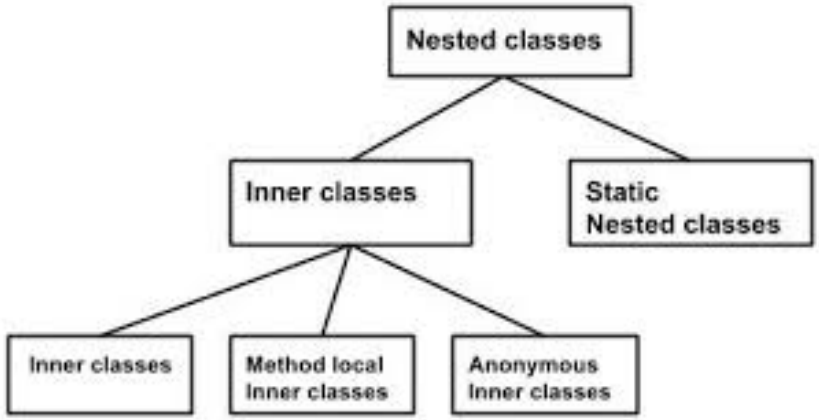
```
public enum myCars{
    HONDA,BMW,TOYOTA
};
```

its optional !!!

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

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

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

Inner classes



- Inner class?
  - A class defined inside another class.
  - Why to define inner classes
- Inner classes
  - Top level inner class
  - Method local inner class
  - Anonymous Inner class
  - Method local argument inner class
- Static inner classes

Top level inner class

- Non static inner class object can't be created without outer class instance
- All the private data of outer class is available to inner class
- Non static inner class can't have static members
  - http://www.avajava.com/tutorials/lessons/iterator-pattern.html

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

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

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

    public static void main(String[] args) {

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

        objectB.foo();
```

Method local inner class

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

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

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

}
```

Anonymous Inner class

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

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

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

What is Exception?

- An exception is an abnormal condition that arises while running a program.

Examples:

- 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...

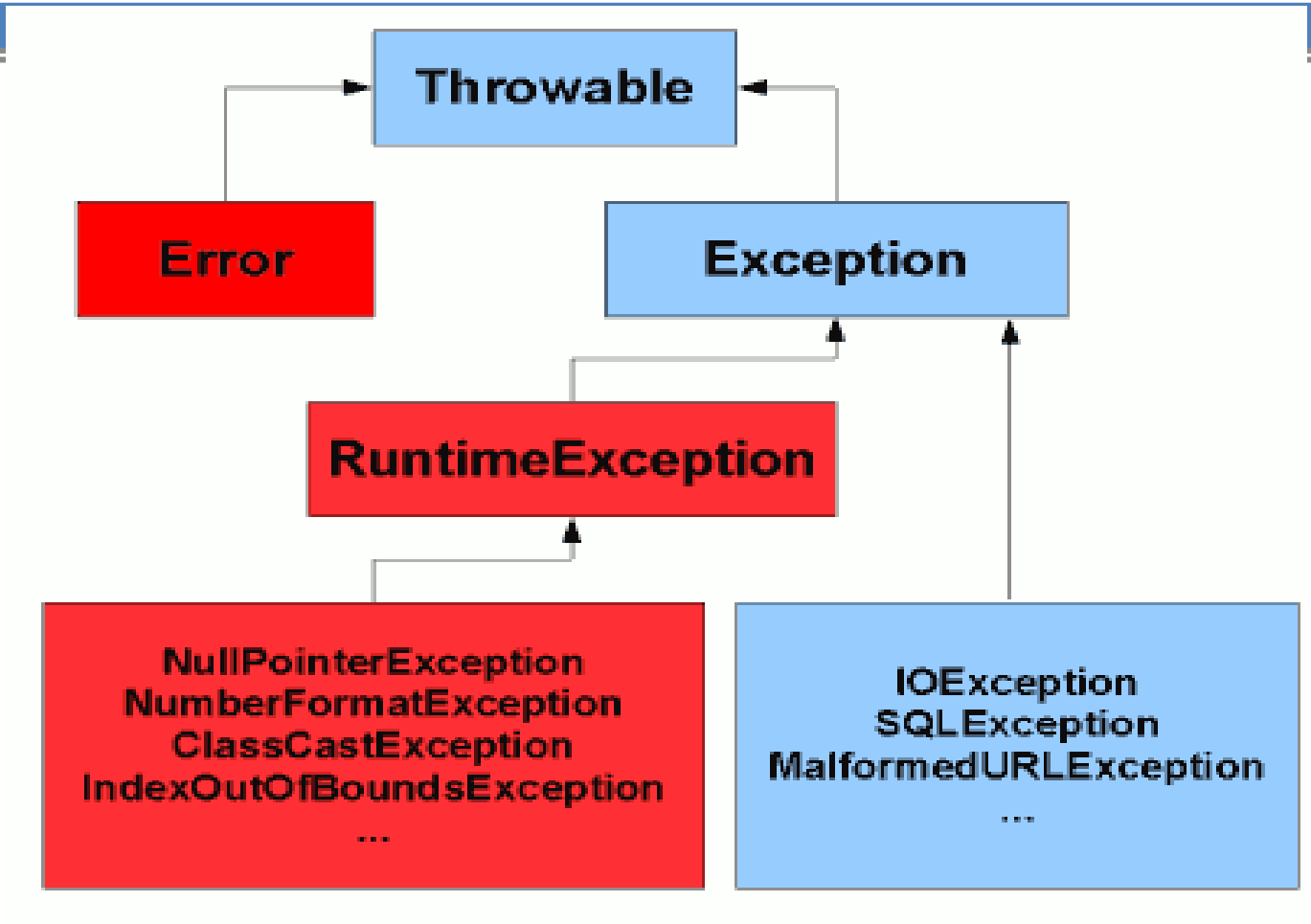
Type of exceptions

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

- 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 key words
- Catch
- Throw

Exception Hierarchy



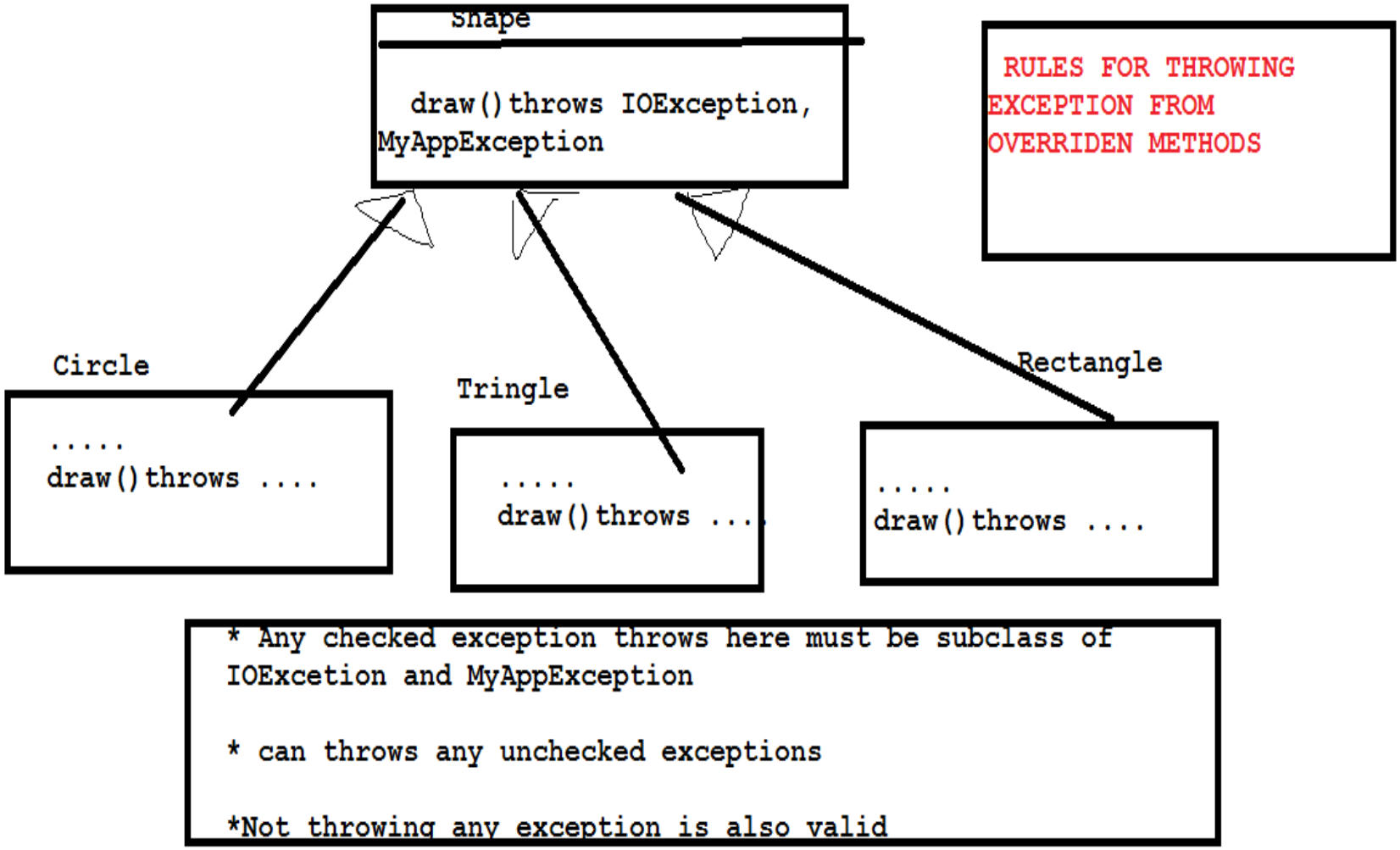
Exceptions and their Handling

- When the JVM encounters an error such as divide by zero, it creates an exception object and throws it – as a notification that an error has occurred.
- If the exception object is not caught and handled properly, the interpreter will display an error and terminate the program.
- If we want the program to continue with execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then take appropriate corrective actions. This task is known as *exception handling*.

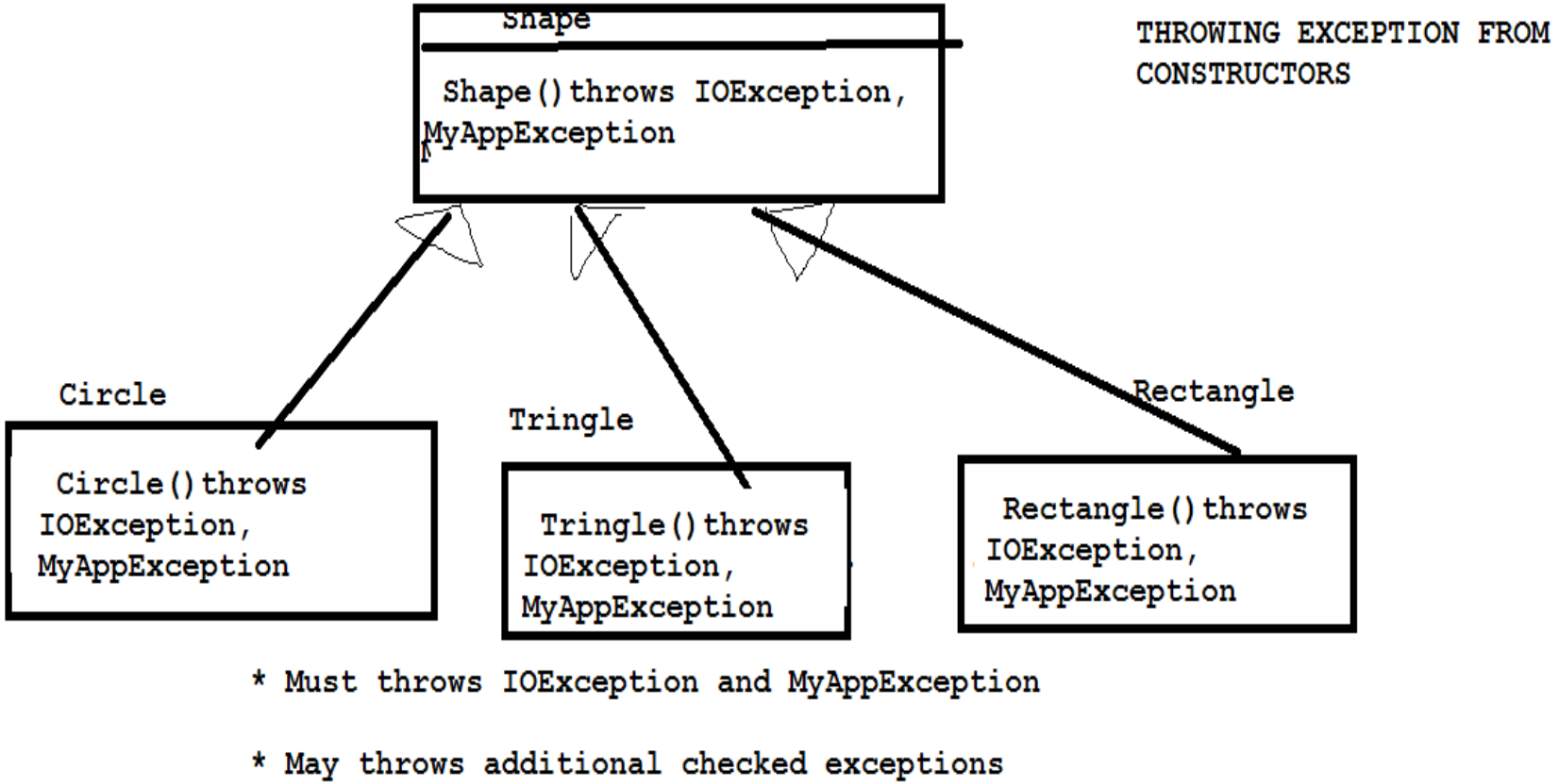
Common Java Exceptions

- 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

Rules for throwing exception from overridden

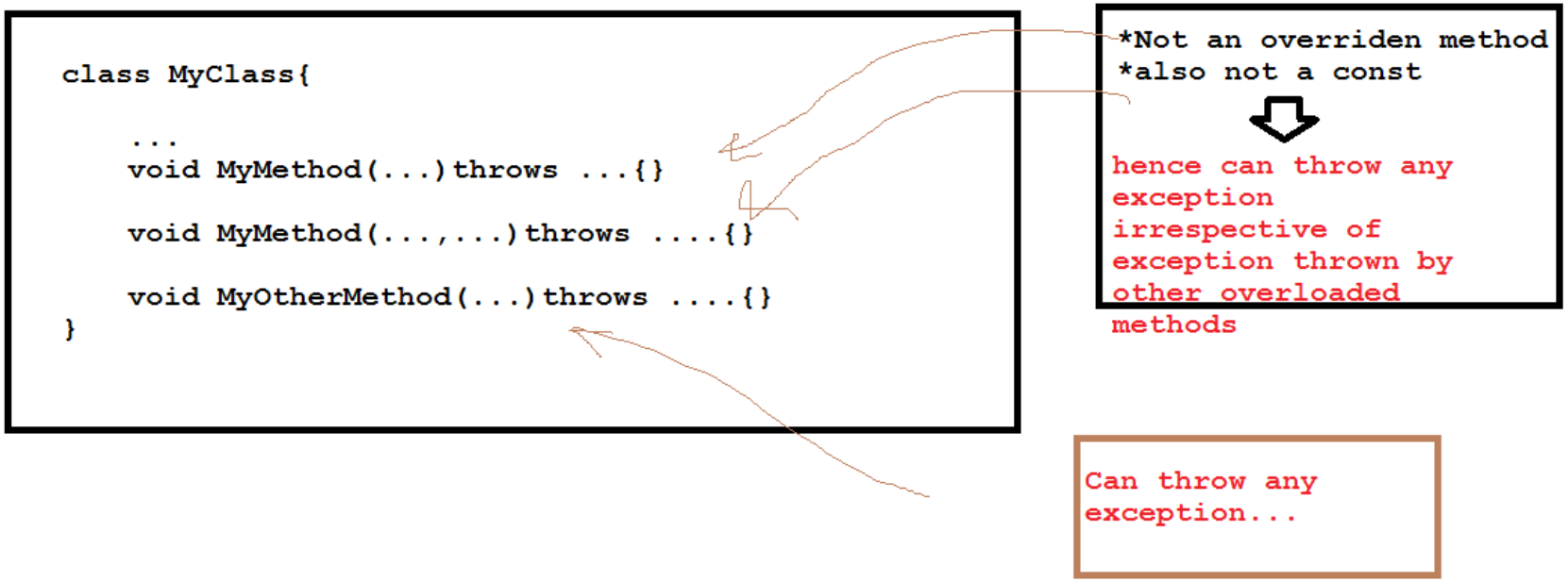


Rules for throwing exception from constructors



Rules for throwing exception from overloaded and other methods

Throwing Exception from Overloaded and other methods...



rgupta.mtech@gmail.com

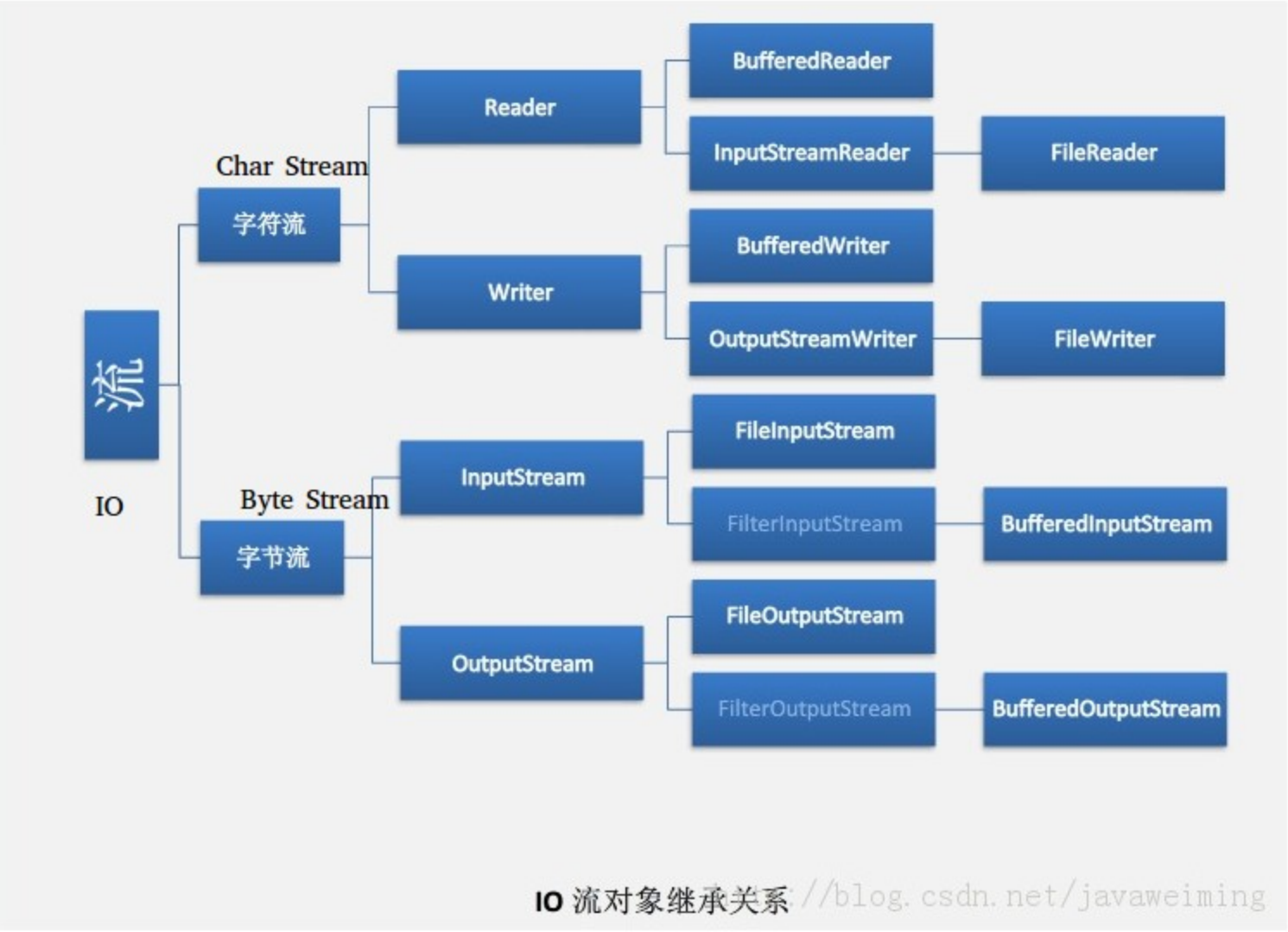
Stream

- 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
    - All byte stream class are like XXXXXXXXStream
  - character: for text data
    - All char stream class are like XXXXXXXXReader/ XXXXXXWriter

Some important classes form java.io

TABLE 6-1 java.io Mini API

java.io Class	Extends From	Key Constructor(s) Arguments	Key Methods
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of Java 5) String (as of Java 5) OutputStream Writer	close() flush() format()*, printf()* print(), println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()
			*Discussed later



IO 流对象继承关系

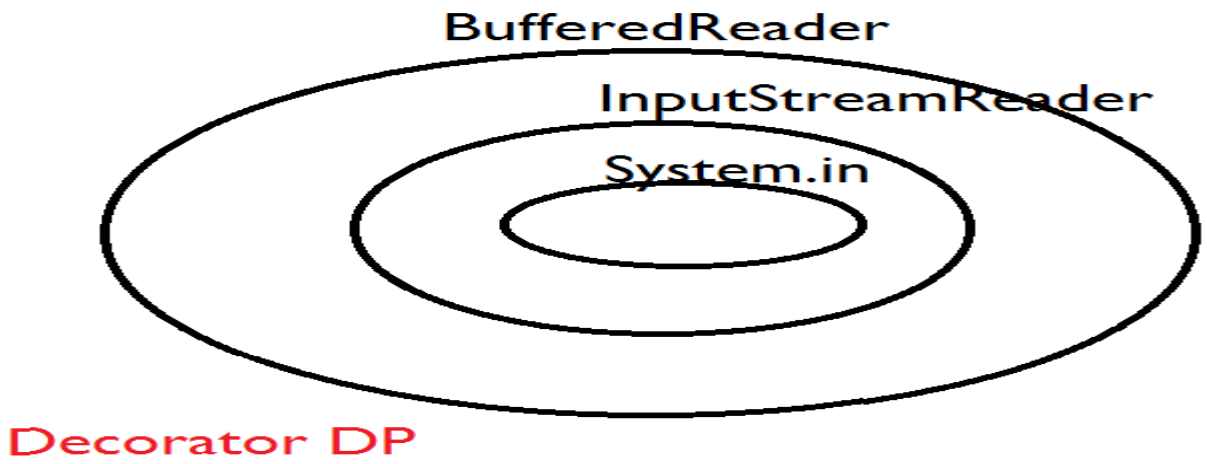
System class in java

- System class defined in java.lang package
- It encapsulate many aspect of JRE

- System class also contain 3 predefine stream variables
  - in
    - System.in (InputStream)
  - out
    - System.out(PrintStream)
  - err
    - System.err(console)

BufferedReader 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")));**



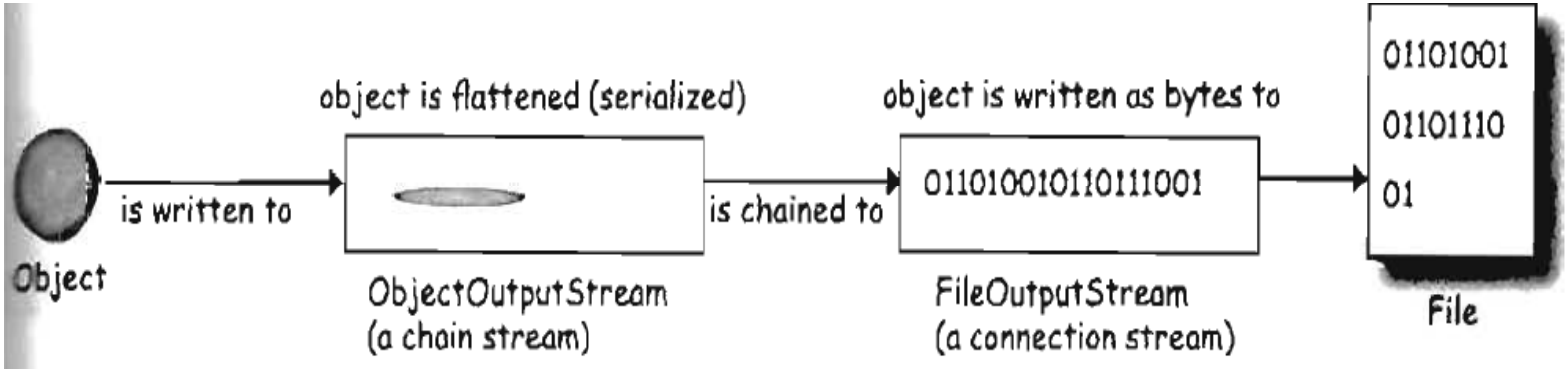
File

- File abstraction that represent file and directories
  - File f=new File("....");
  - boolean flag= file.createNewFile();
  - boolean flag= file.mkDir();
  - boolean flag=file.exists();
- FileWriter

```
File file = new File( "fileWrite2.txt");
FileWriter fw =new FileWriter(file);
fw.write("howdy\nfolks\n"); // write characters to
fw.flush(); // flush before closing
fw.close(); // close file when done
```

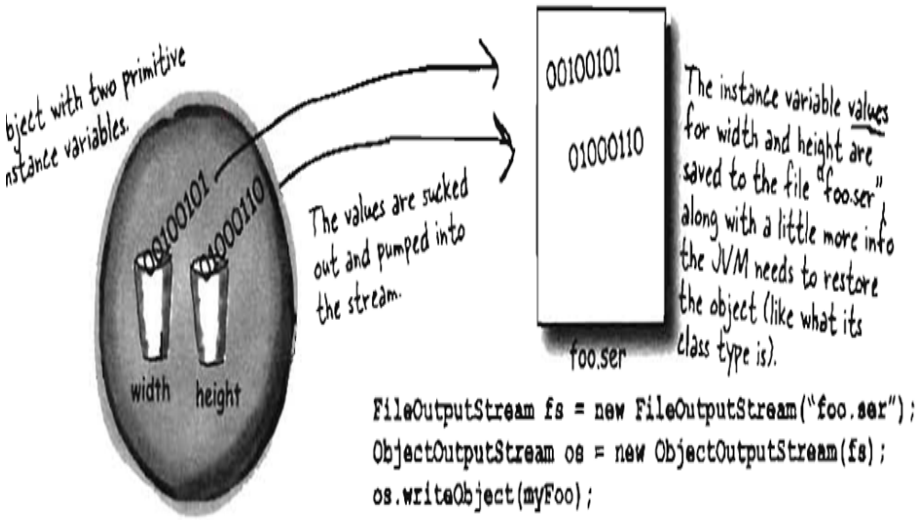
Serialization

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



What actually happens during Serialization

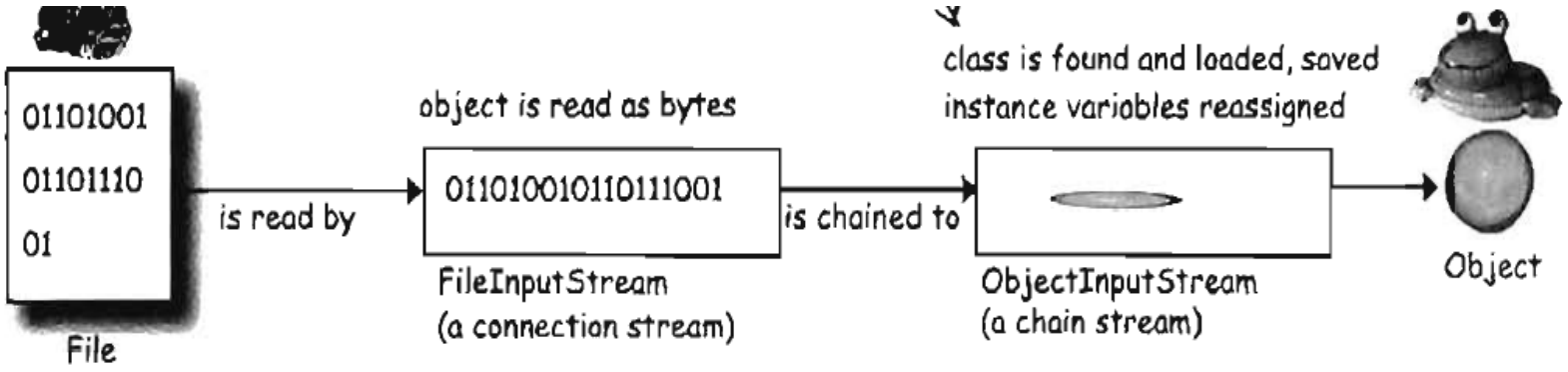
- The state of the object from heap is sucked and written along with some meta data in an file….



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

De- Serialization

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



Hello world Example...

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

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

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

}
```

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

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

os.writeObject(box);
```

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

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

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

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

Differences between Serialization and Externalization?

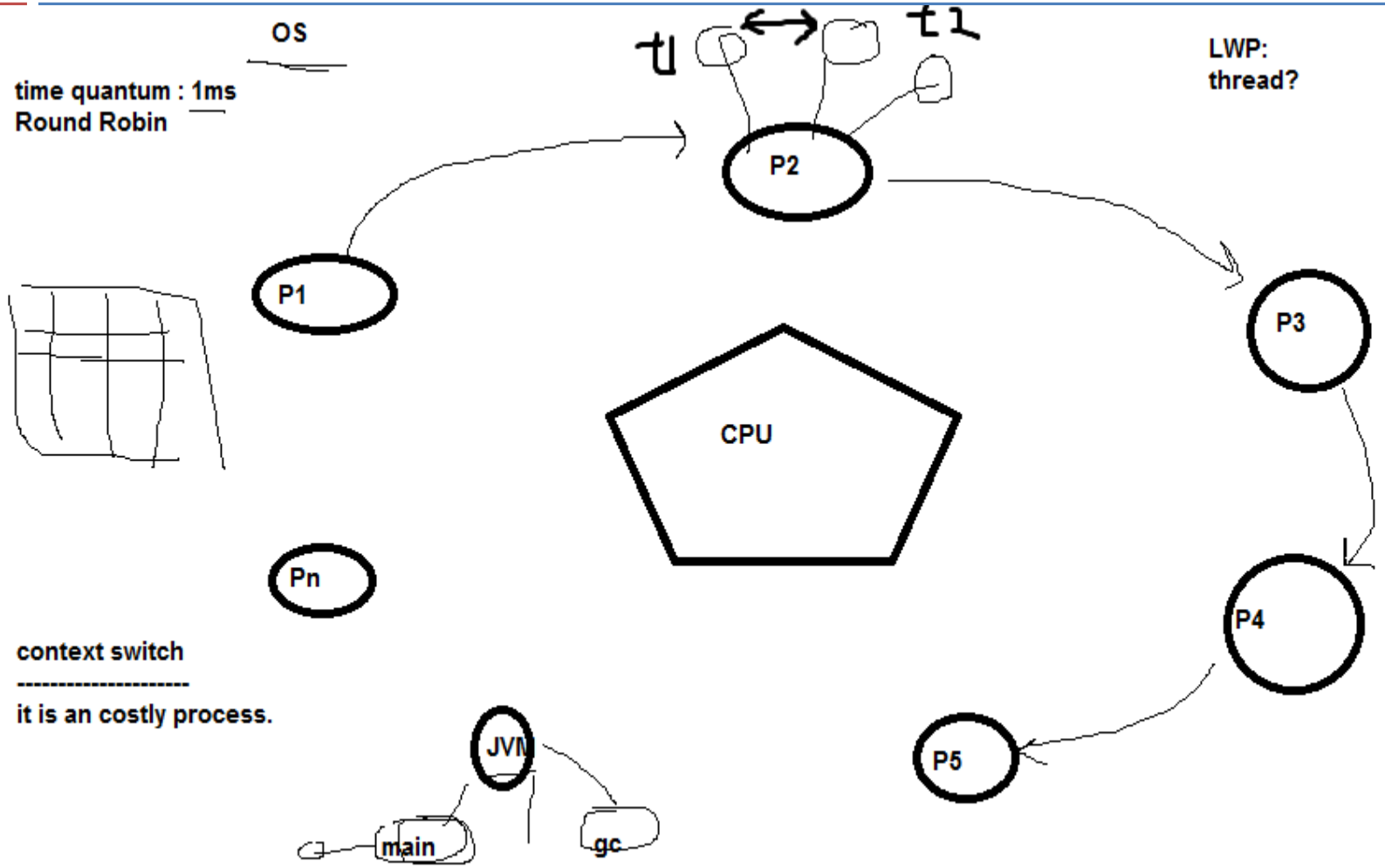
Serialization	Externalization
It is Meant for Default Serialization.	It is Meant for Customized Serialization.
Here Everything Takes Care by JVM and Programmer doesn't have any Control.	Here Everything Takes Care by Programmer and JVM doesn't have any Control.
In Serialization Total Object will be Saved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


DAY 5

Day 5: Introduction to Java threads, thread life cycle, synchronization, dead lock

- Program vs process
- Thread as LWP
- Creating and running thread
- Thread life cycle
- Need of synchronization
- Producer consumer problem
- dead lock
- Hands on & Lab

What is threads? LWP



rgupta.mtech@gmail.com

Basic fundamentals

- Thread: class in java.lang
- thread: separate thread of execution

What happens during multithreading?

1. JVM calls main() method
2. main() starts a new thread. Main thread is tempory frozen while new thread start running so JVM switch between created thread and main thread till both complets

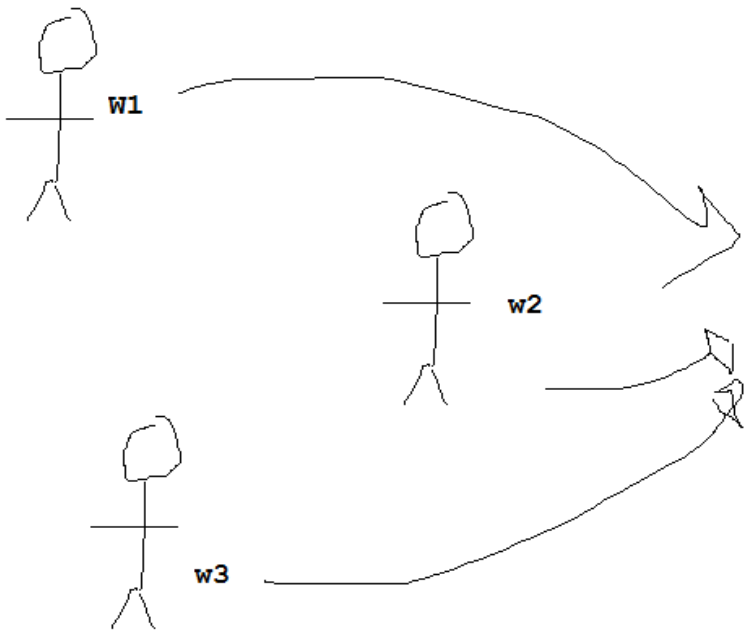
Creating threads Java?

- Implements Runnable interface
- Extending Thread class……..

- Job and Worker analogy…

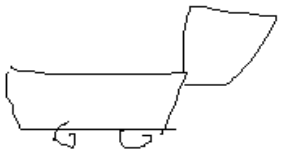
```
class Job implements Runnable{


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

}
```

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



Thread
----
consider object of threads as worker

Implementation of Runnable as Job



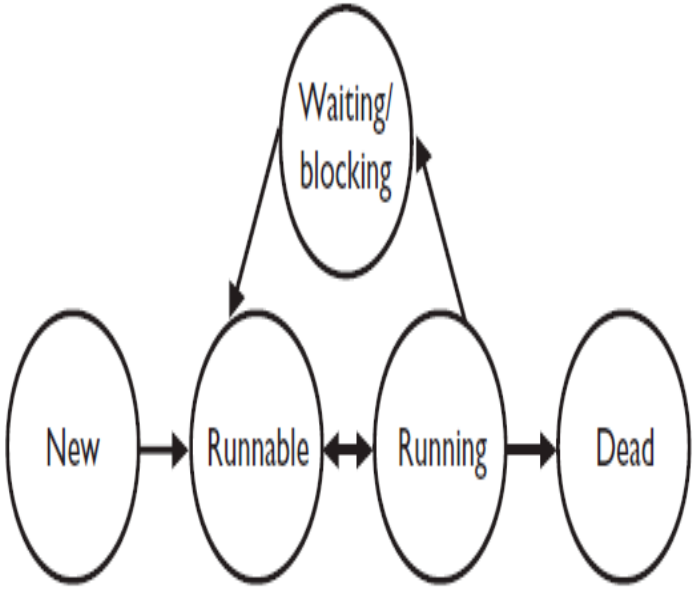
Job

simulation

Sleep()

Thread life cycle...

- If a thread is blocked (put to sleep) specified no of ms should expire
- If thread waiting for IO that must be over..
- If a threadcalls wait() then another thread must call notify() or notifyAll()
- If an thred is suspended() some one must call resume()



Java.lang.Thread

- Thread()
  - construct new thread
- void run()
  - must be overriden
- void start()
  - start thread call run method
- static void sleep(long ms)
  - put currently executing thread to sleep for specified no of millisecond
- boolean isAlive()
  - return true if thread is started but not expired
- void stop()
- void suspend() and void resume()
  - Suspend thread execution….

Java.lang.Thread

- void join(long ms)
  - Main thread Wait for specified thread to complete or till specified time is not over
- void join()
  - Main thread Wait for specified thread to complete
- static boolean interrupted()
- boolean isInterrupted()
- void interrupt()
  - Send interrupt request to a thread

Creating Threads

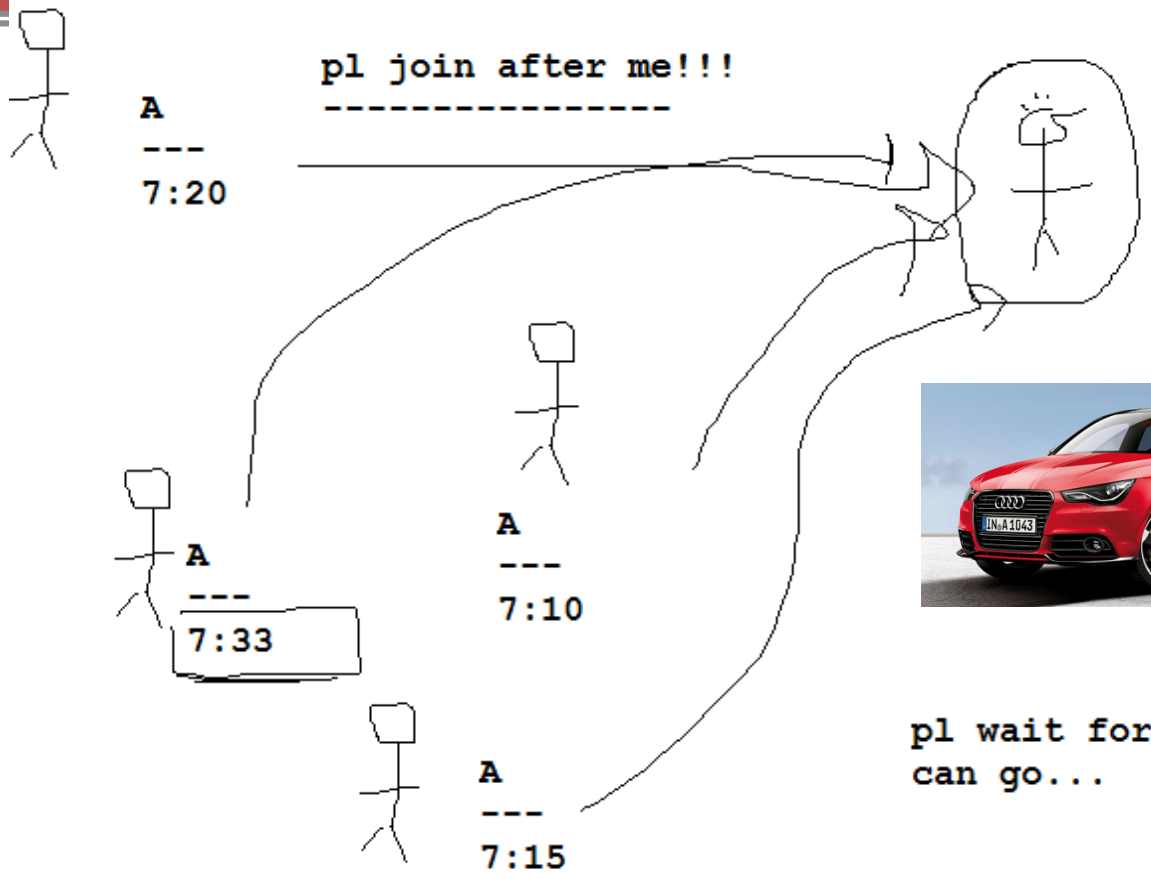
- By extending Thread class

```
class MyThread extends Thread
{
    @Override
    public void run()
    {

    }

}
```

- Implementing the Runnable interface

```
class MyRunnable implements Runnable
{

    @Override
    public void run()
    {

    }

}
```

Understanding join() method



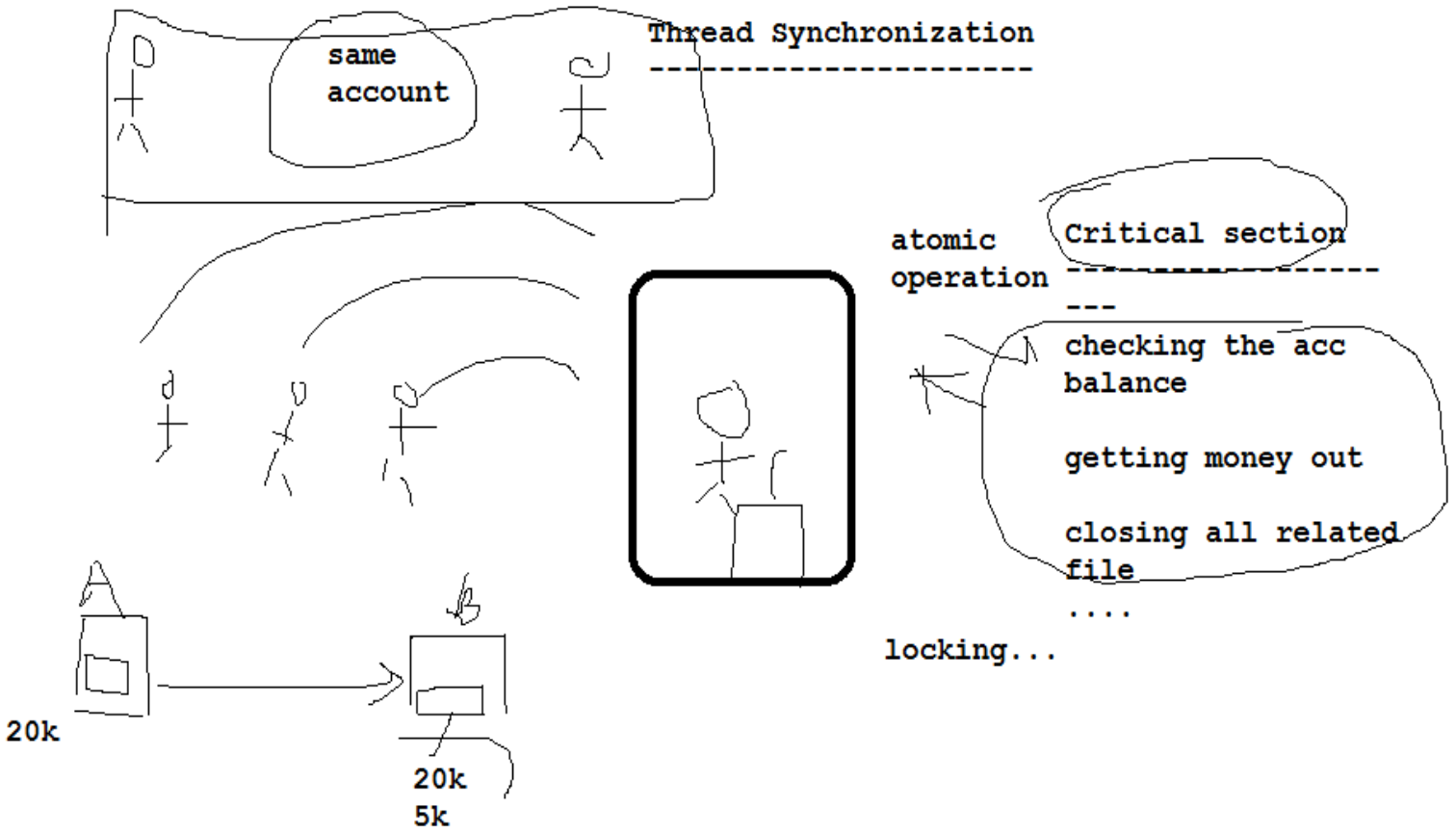
pl wait for me for 5 min else u can go...

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

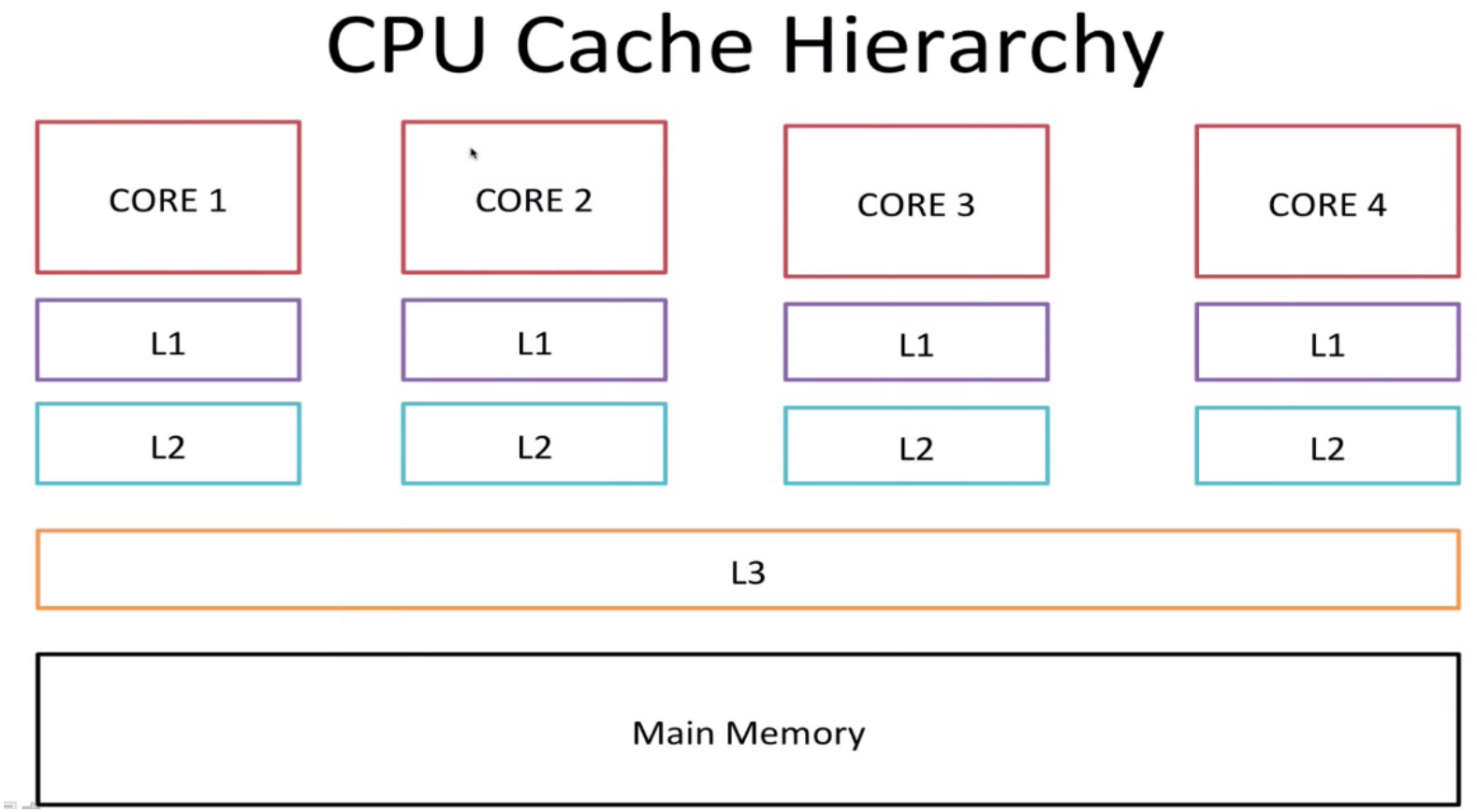


Synchronization

- 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.

volatile


CPU Cache Hierarchy
CORE 1
CORE 2
CORE 3
CORE 4
L1
L1
L1
L1
L2
L2
L2
L2
L3
Main Memory

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");
```

Inter thread communication

- Java have elegant Interprocess communication using wait() notify() and notifyAll() methods
- All these method defined final in the Object class
- Can be only called from a synchronized context

wait() and notify(), notifyAll()

- wait()
  - Tells the calling thread to give up the monitor and go to the sleep until some other thread enter the same monitor and call notify()
- notify()
  - Wakes up the first thread that called wait() on same object
- notifyAll()
  - Wakes up all the thread that called wait() on same object, highest priority thread is going to run first

Incorrect implementation of produce consumer

```
class Q
{
        int n;
        synchronized int get()
        {
            System.out.println("got:"+n);
            return n;
        }
        synchronized void put(int n)
        {
            this.n=n;
            System.out.println("Put:"+n);
        }
}
```

```
class Producer implements Runnable
{
    Q q;
    public Producer(Q q) {
        this.q=q;
        new Thread(this,"Producer").start();
    }
    public void run()
    {
        int i=0;
        while(true)
            q.put(i++);
    }
}
```

```
public class PandC {
public static void main(String[] args) {
    Q q=new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("ctrol C for exit");
}
}
```

```
class Consumer implements Runnable
{
    Q q;
    Consumer(Q q)
    {
        this.q=q;
        new Thread(this,"consumer").start();
    }
    public void run()
    {
        while(true)
            q.get();
    }
}
```

Correct implementation of produce consumer ...

```
class Q
{   int n;
        boolean valueSet=false;
        synchronized int get()
        {
            if(!valueSet)
                try
                {
                    wait();
                }
            catch(InterruptedException ex){}
            System.out.println("got:"+n);
            valueSet=false;
            notify();
            return n;
        }
        synchronized void put(int n)
        {
            if(valueSet)
                try
                {
                    wait();
                }
            catch(InterruptedException ex){}
            this.n=n;
            valueSet=true;
            System.out.println("Put:"+n);
            notify();
        }
}
```

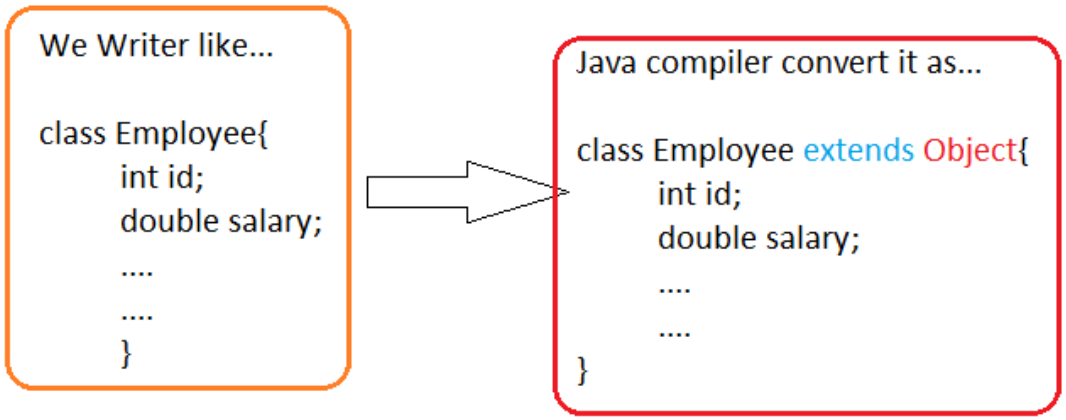

DAY
06

Day 6:Java Collection, Generics

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

Object

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



- Why Java has provided this class?

Method defined in Object class...

- String toString()
- boolean equals()
- int hashCode()
- clone()
- void finalize()
- getClass()
- Method that can’t be overridden
  - final void notify()
  - final void notifyAll()
  - final void wait()

toString( )

- If we do not override toString() method of Object class it print Object Identification number by default
- We can override it to print some useful information....

```
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

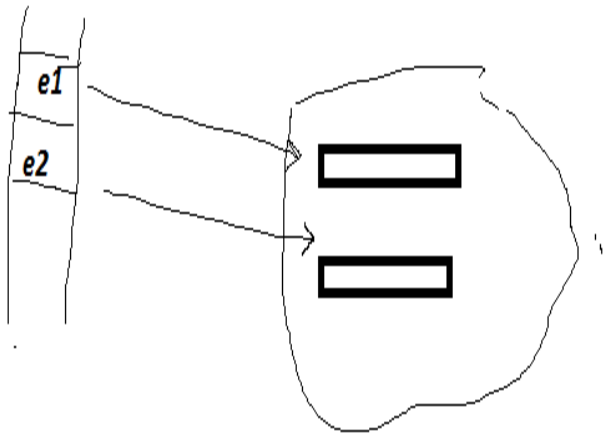
toString()

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

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

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

}
public class DemoToString {
     public static void main(String[] args) {
          Employee e=new Employee(22, 333333.5);
          System.out.println(e);
     }
}
```

equals

- What O/P do you expect in this case……

```
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
```



- 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()

- Don't forget DRY run.....

```
@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;
    }
```

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;
    }
```

clone()

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

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

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

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


}
```

finalize()

- As you are aware ..Java don’t support destructor
- Programmer is free from memory management
- Memory mgt is done by an component of JVM ie called Garbage collector GC
- GC runs as low priority thread..
- We can override finalize() to request java
  “Please run this code before recycling this object”
- Cleanup code can be written in finalize() method
- Not reliable, better not to use…
- Demo programm
  - WAP to count total number of employee object in the memory at any moment of time if an object is nullified then reduce count….

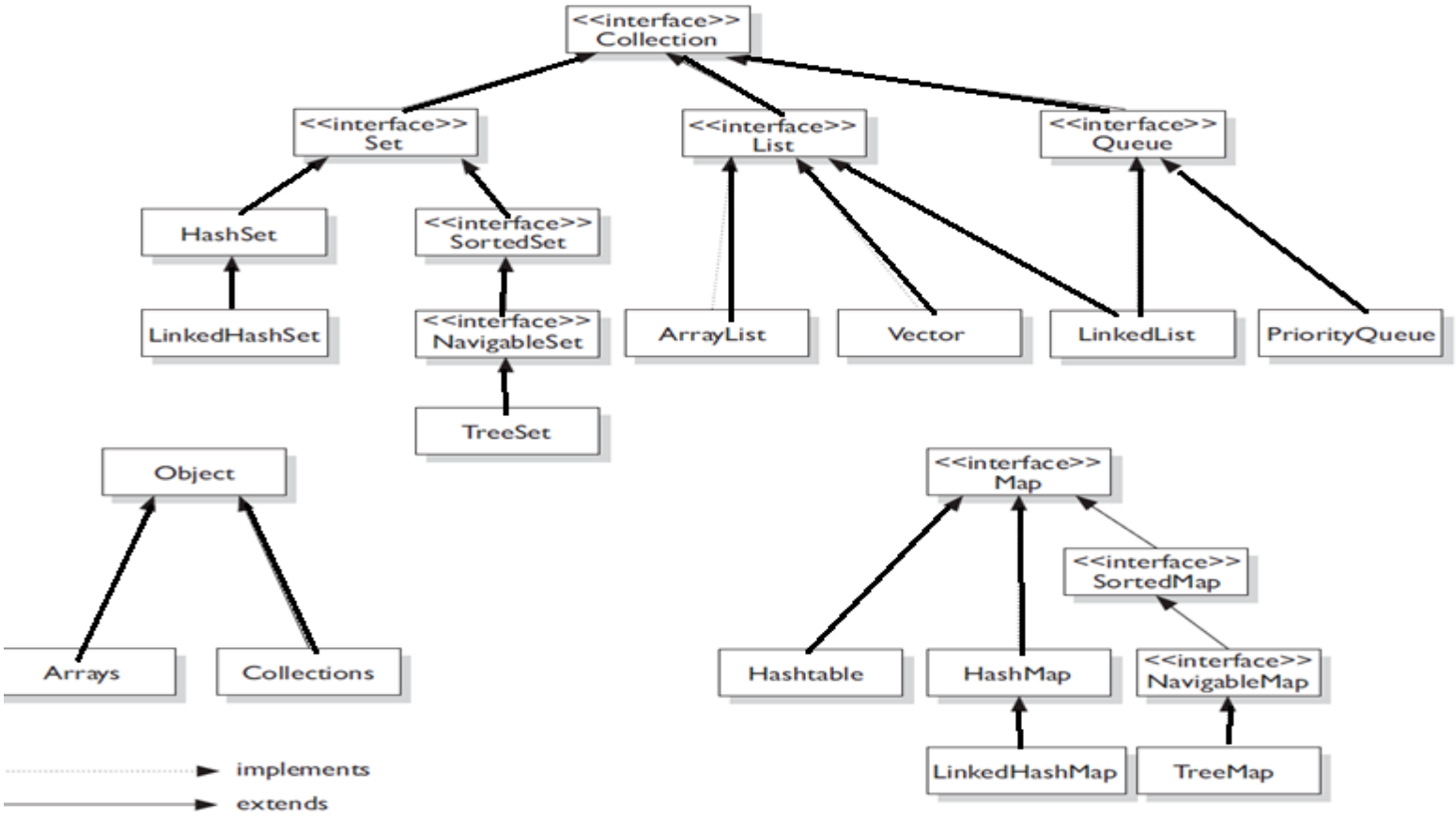
Java collection

- 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

collection

- collection
  - Why it is called an framework?
  - Readymade Data structure in Java...



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.

ArrayList: aka growable

```
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);
    }

}
```

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

rgupta.mtech@gmail.com

Useful examples...

user define funtion to print the arraylist/linkedlist

```
----------------------------------------------------------

printMe(list1);
....
.....


public void printMe(list<String>list){

        for(String s:list)
        {
                Sysout(s);
        }
}
```

user define funtion to remove stuff from a arraylist /linkedlist

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


...
...

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

rgupta.mtech@gmail.com

Useful examples...

Merging two link lists

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

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

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

Removing every second element from an linkedList

```
itb=b.iterator();

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

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

                itb.remove();
        }
}
```

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

Useful examples...

```
user define funtion to print linkedlist in reverse order
-----------------------------------------------------------

reversePrint(list);
...
....

public void reversePrint(list<String>l ){

        ListIterator<String>it=l.iterator(l.size());

        while(it.hasPrevious())
                Sysout(it.previous());
}
```

rgupta.mtech@gmail.com

fundamental diff bw ArrayList and LinkedList

- ArrayLists manage arrays internally.
  [0][1][2][3][4][5] ....
- List<Integer> arrayList = new ArrayList<Integer>();

- LinkedLists consists of elements where each element has a reference to the previous and next element
  [0]->[1]->[2] ....
  <- <-

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 ArrayList 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
```

rguptaiiitcon@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");
```

- Getting value
  - String text = map.get(6);
  - System.out.println(text);

Looping through HashMap

```
for(Integer key: map.keySet())
      {
         String value = map.get(key);

         System.out.println(key + ": " + value);
       }
```

most imp thing to remember

---

order of getting key value is not maintained

ie hashMap dont keep key and value in any particular order

Other map varients

- LinkedHashMap
  - Aka. Doubly link list
  - key and value are in same order in which you have inserted.....
- TreeMap
  - sort keys in natural order(what is natural order?)
  - for int
    - 1,2,3.....
  - for string
    - "a","b".....
  - For user define key
    - Define sorting order using Comparable /Comparator
-

set

- Don’t allow duplicate element

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>();
```

Printing freq of unique words
from a file in increseing order of freq

---

words	freq
Apple	7
Ball	5
...	

rgupta.mtech@gmail.com

User define key in HashMap

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

HashMap vs Hashtable

- Hashtable is threadsafe, slow as compared to HashMap
- Better to use HashMap

- Some more intresting difference
  - Hashtable give runtime exception if key is “null” while HashMap don’t

Generics

- Before Java 1.5
  - List 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…

Issues with Generics

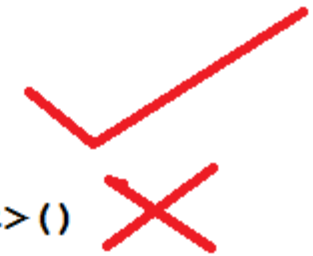
- Try not to mix non Generics code and Generics code…we can have strange behaviour.

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

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

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

        strangMethod(list);

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

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

}
```

Polymorphic behaviour

```
class Animal {
}

class Cat extends Animal{
}

class Dog extends Animal{
     }
```

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

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



rgupta.mtech@gmail.com

<? extends XXXXXX>

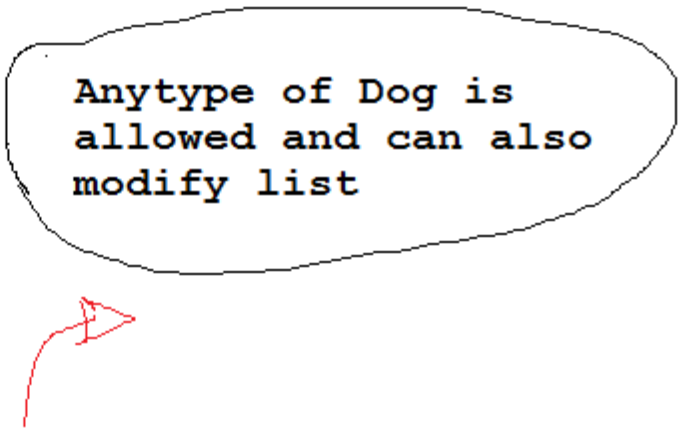
```
package com;
import java.util.*;

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

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

        strangMethod(list);

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

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

}
```

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

rgupta.mtech@gmail.com

<? Super XXXXX>

```
class Animal {
}

class Cat extends Animal{
}

class Dog extends Animal{
     }

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

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

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



rgupta.mtech@gmail.com

Generic class

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

     public T getMyObject() {
          return myObject;
     }

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

}
public class GenClass {

          public static void main(String[] args) {
               MyObject<String> o=new MyObject<String>();
               o.setMyObject(new Integer(22));
               //System.out.println(it.intValue());
          }
}
```

will not compile !!!

rgupta.mtech@gmail.com

Generic method

```
class MaxOfThree{

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

}
```


Day
7

Day-7: Annotation, Java Reflection

- Introduction to java reflection
- Java Annotation, JDK annotation, creating custom annotation, Annotation and reflection
- Introduction to Design pattern
- Hands on & Lab

Java Reflection

Java Reflection:

=> Reflection is aka class manipulator?
=> Used to manipulate classes and everything in a class
=> Can slow down a program because the JVM can not optimize the code
=> Can not used with applets
=> Should be used spraringly
What is Java reflection?

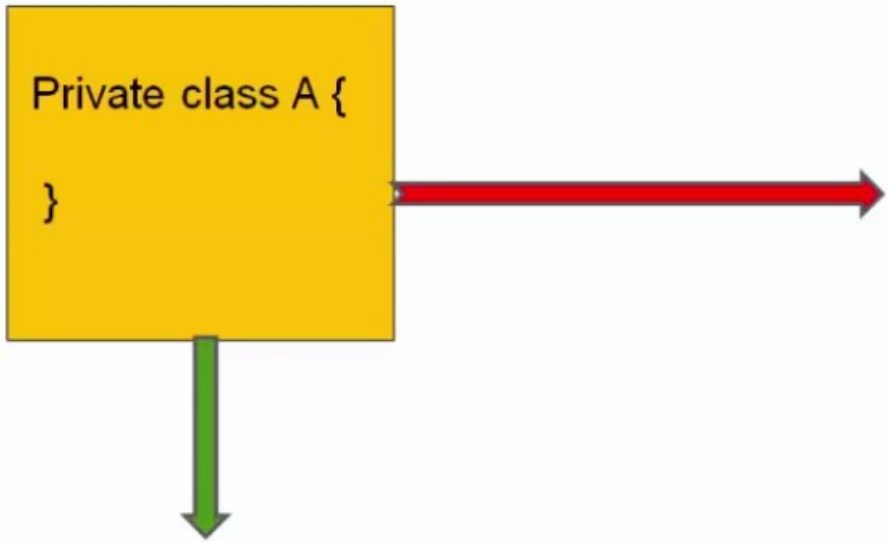
"Reflection is the process of analysing the capabilities of a class at runtime"

That is analysiing the details about the class which include the methods in the class, the variables of the class, the constructors in the class, the interface that the class is implementing, the methods that are coming from the interface
To gather all the above details is provided by reflection api

Reflection api is not the project development but used for PRODUCT development
Example application area:
JVM development, server design, framework design, tool design, compiler

rgupta.mtech@gmail.com

Java Reflection



COMPILER:

The compiler will read this class according to the CLASS RULES defined in the compiler.

CLASS RULES
Can have only PUBLIC and DEFAULT access modifier in outer class. Will not allow PRIVATE OR PROTECTED access modifier for outer class This rule is analysed by the REFLECTION API present in the compiler and it shows an error "MODIFIER PRIVATE not allowed"

Similarly rules for methods in the class, constructors in the class, interfaces inherited by the class, super classes extended by the class are also present and their access modifiers are also checked by REFLECTION API present in the compiler

Comment: An outer class cannot have a PRIVATE declaration. An inner class can have PRIVATE declaration. A class can be declared as PUBLIC and DEFAULT; it cannot HAVE PRIVATE or PROTECTED access modifier

rgupta.mtech@gmail.com

Java Reflection

Reflection Package

java.lang.reflect package provides classes for performing reflection api.

Some classes in this reflect packages are :

java.lang.Class :
base class to perform reflection api.
This is used togather to get metadata information about class

java.lang.reflect.Field:
Complete declerative information of a particular variable ie.
metadata of a particular variable like access modifier, datatype,
value and name of variable etc

java.lang.reflect.Method:
complete declaritive information about a method ie. is able to store metadata
about methods, method names, access modifier, return type, parameters in method,
exception thrown etc

java.lang.reflect.Constructor:
information about constructs like name of constructors, access modifier, parameter type

java.lang.reflect.Modifier:
complete metadata about access modifier

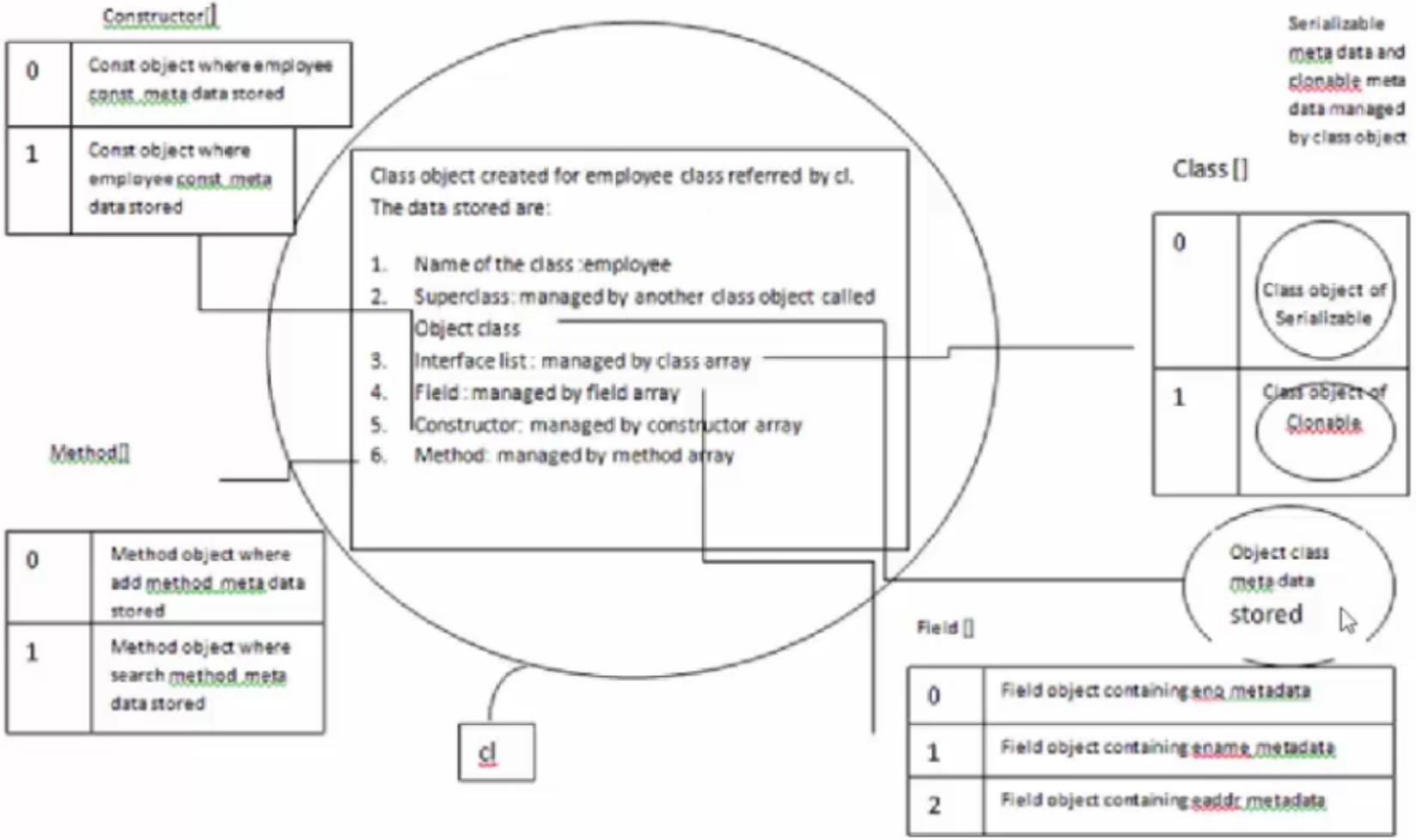
rgupta.mtech@gmail.com

Java Reflection

```
public class Employee implements Serializable, Cloneable{
    // variable ==4
    public int empNo;
    public static int companyName;
    public String name;
    public final String add="delhi";

    // ctr==2
    public Employee(int empNo, String name) {
        this.empNo = empNo;
        this.name = name;
    }
    public Employee() {}

    // methods==2
    public void add(int id, String name,String address){

    }
    public Employee search(int id){return new Employee();}

}
```

rgupta.mtech@gmail.com

Java Reflection



rgupta.mtech@gmail.com

Changing behaviour at runtime

Anyway, reflection does not allow you to change code behaviour, it can only explore current code, invoke methods and constuctors, change fields values, that kind of things.

If you want to actually change the behaviour of a method you would have to use a bytecode manipulation library such as ASM. But this will not be very easy, probably not a good idea...

Patterns that might help you :

- If the class is not final and you can modify the clients, extend the existing class and overload the method, with your desired behaviour. Edit : that would work only if the method were not static !
- Aspect programming : add interceptors to the method using AspectJ

Anyway, the most logical thing to do would be to find a way to modify the existing class, work-arounds will just make your code more complicated and harder to maintain.

Annotation

Annotations starting from 5.0.

This feature was added to Java 5.0 as a result of the JSR 175 namely "A Metadata Facility for the JavaTM Programming Language".

Built-in Annotations in Java

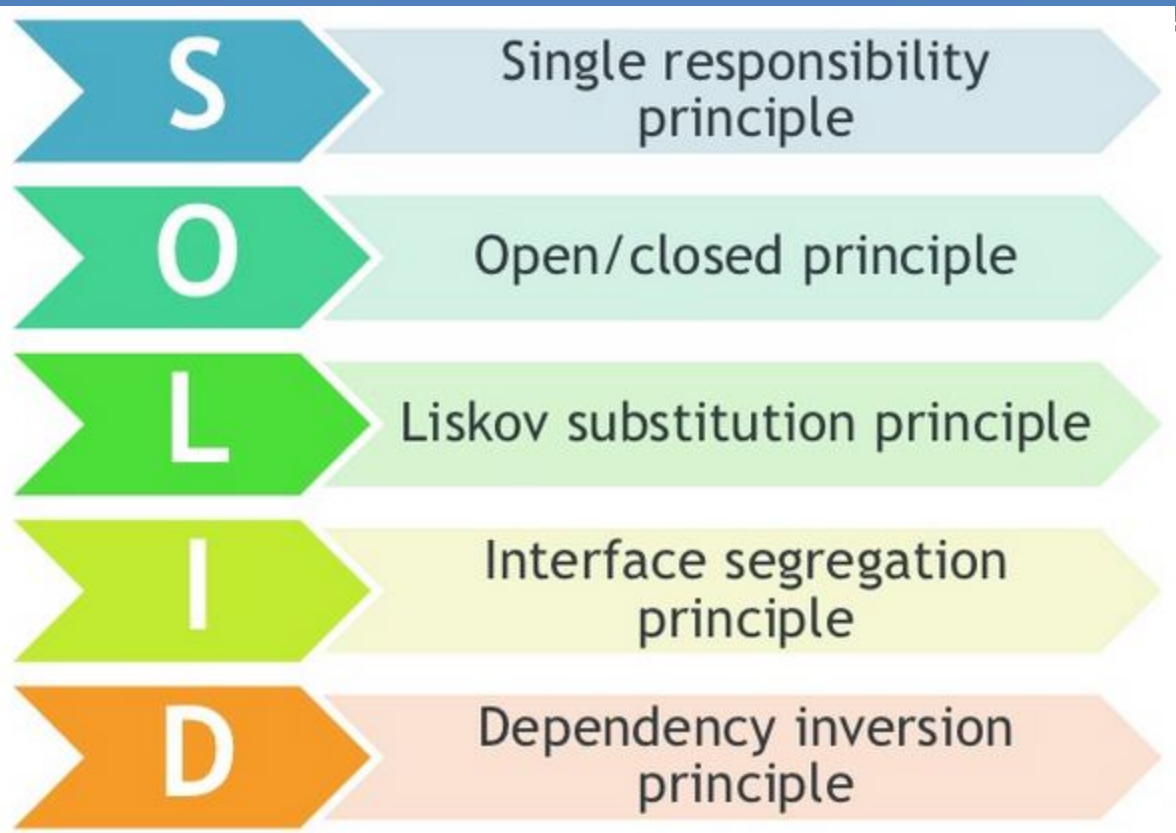
@Override
@Deprecated
@SuppressWarnings
@Target
@Retention
@FunctionalInterface

User-defined Annotations

```
Target({ElementType.FIELD, ElementType.LOCAL_VARIABLE})
public @interface Persistable{
        String fileName() default "defaultMovies.txt";
}
```

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


S
Single responsibility principle
O
Open/closed principle
L
Liskov substitution principle
I
Interface segregation principle
D
Dependency inversion principle


Day
8

Day 8: GOF design patterns

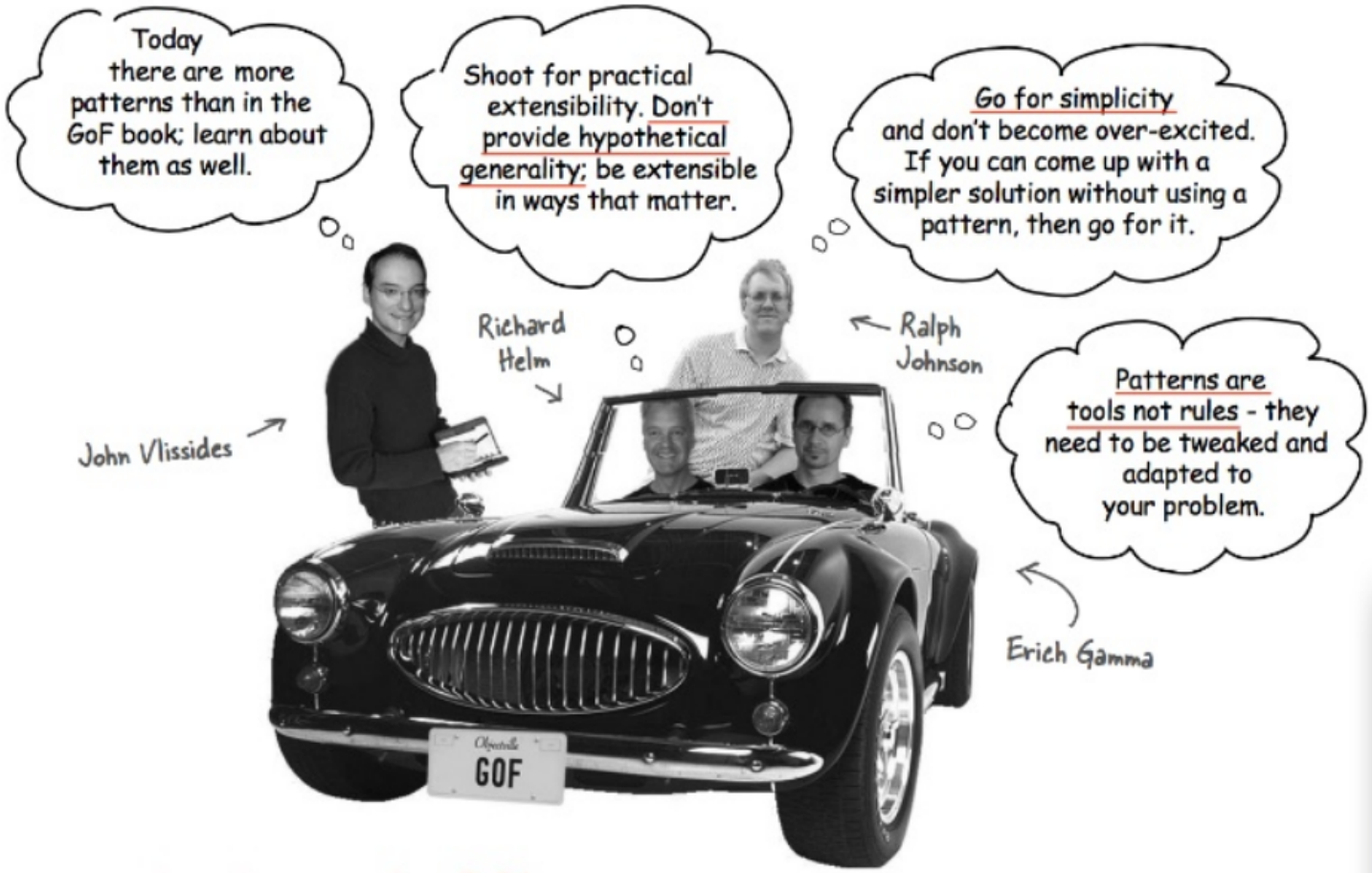
- Pattern categories: Creational, Structural, Behavioral
- Creational Patterns: Singleton, Factory, Builder, Prototype
- Structural Patterns: Decorator, Facade Pattern, Proxy
- Behavioral Patterns: Iterator, Observer, Strategy,Template Method
- Hands on & Lab

Design pattern

- Proven way of doing things
- **Gang of 4 design patterns ???**
- **total 23 patterns**
- **Classification patterns**
  1. **Creational**
  2. **Structural**
  3. **Behavioral**

"In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations."

WIKIPEDIA
The Free Encyclopedia



Keep it simple (KISS)

Pattern

- **Context**
  - a design situation giving rise to a design problem
- **Problem**
  - a set of **forces** occuring in that context
- **Solution**
  - a form or rule that can be applied to **resolve** these forces

IF you find yourself in **CONTEXT**
for example **EXAMPLES**,
with **PROBLEM**,
entailing **FORCES**
THEN for some **REASONS**,
apply **DESIGN FORM AND/OR RULE**
to construct **SOLUTION**
leading to **NEW CONTEXT & OTHER PATTERNS**

Design Patterns – Classification

Structural Patterns	Creational Patterns	Behavioral Patterns
• 1. Decorator • 2. Proxy • 3. Bridge • 4. Composite • 5. Flyweight • 6. Adapter • 7. Facade	• 1. Prototype • 2. Factory Method • 3. Singleton • 4. Abstract Factory • 5. Builder	• 1. Strategy • 2. State • 3. TemplateMethod • 4. Chain of Responsibility • 5. Command • 6. Iterator • 7. Mediator • 8. Observer • 9. Visitor • 10. Interpreter • 11. Memento

Patterns classification

- Creational patterns?
  - What is the best way to create object, new is not the best option
- structural patterns
  - Structural Patterns describe how objects and classes can be combined to form larger structures
- Behavioral Patterns
  - Behavioral patterns are those which are concerned with interactions between the objects (talking to each other still loosely coupled)

Creatioal Patterns

Creational Patterns

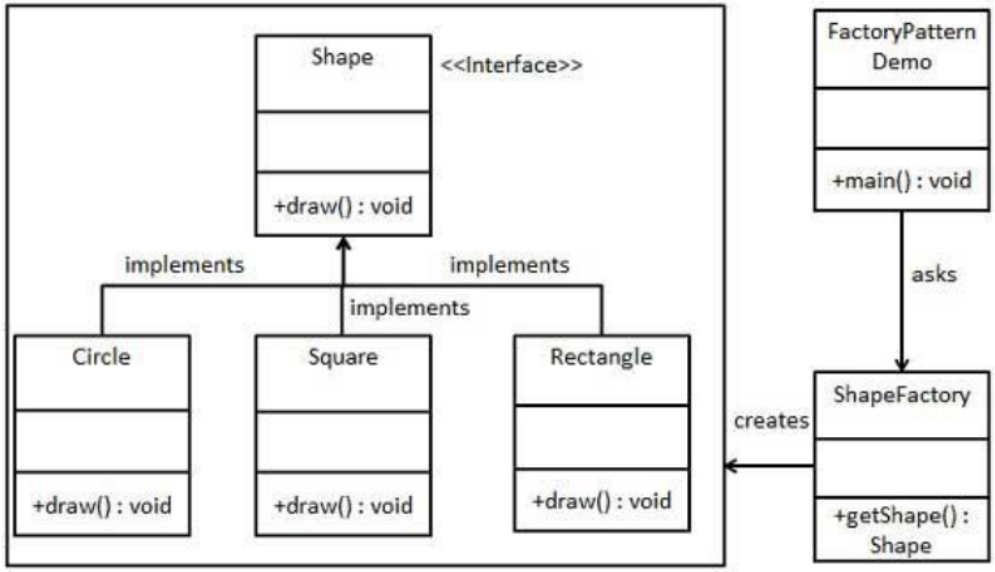
Book book = new Book ();

All the Creational patterns define the best possible way in which an object can be instantiated.

- The new Operator creates the instance of an object, but this is hard-coding.

- we can make use of Creational Patterns to give this a more general and flexible approach.

Simple Factory



-
- **Factory of what? of classes. ..**

In simple words,

“if we have a super class and an sub-classes, and based on data provided, we have to return the object of one of the sub-classes, we use a simple factory”

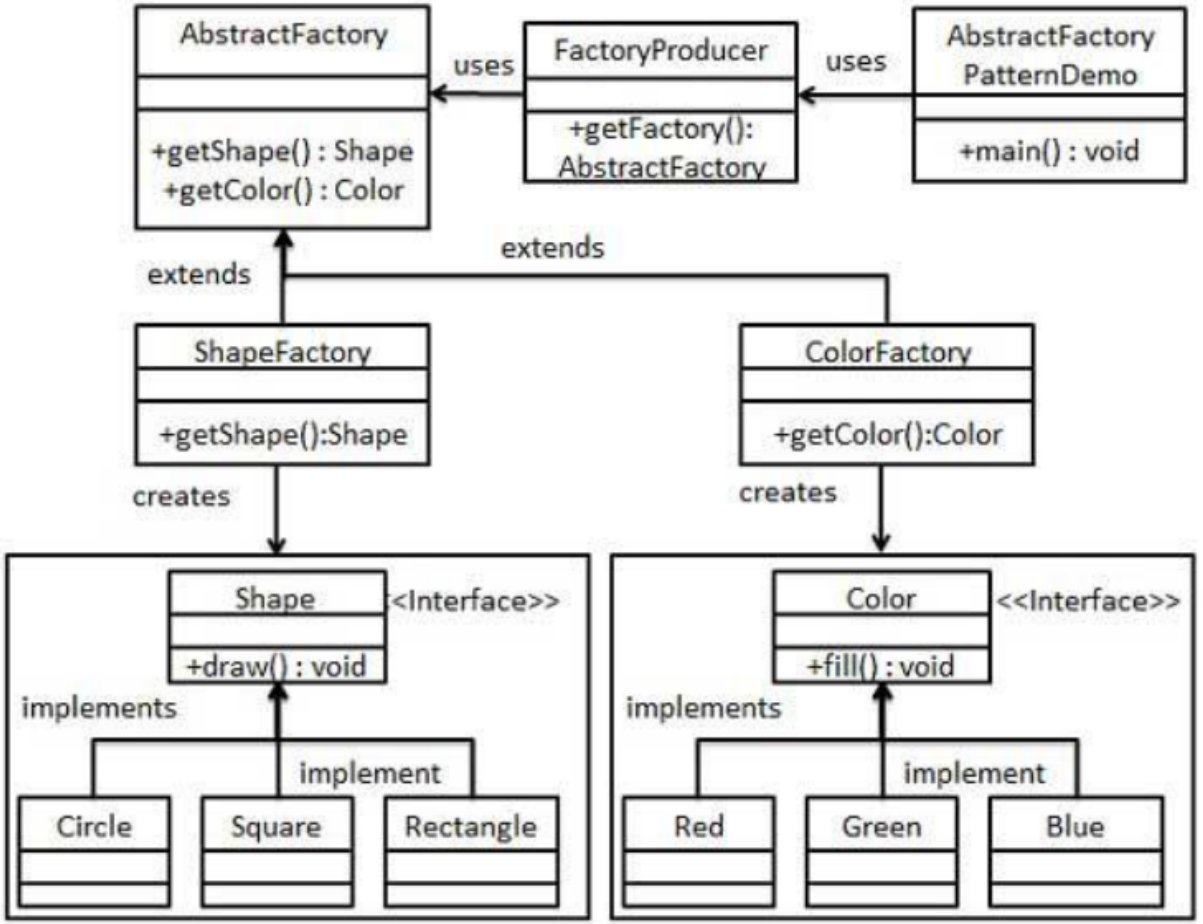
Factory Method Design Pattern

Define an **interface** for creating an object, but let **subclasses** decide which object to **instantiate**. Factory Method lets a class defer instantiation to subclasses.

Abstract Factory Pattern

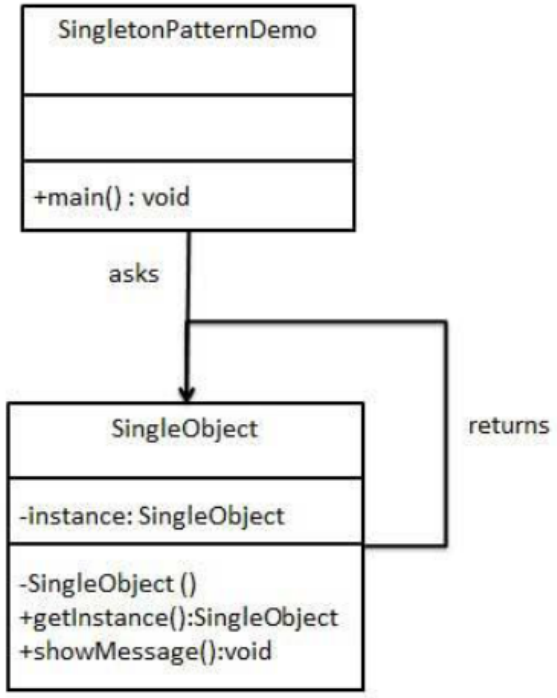
- This pattern is one level of abstraction higher than factory pattern. This means that the abstract factory returns the factory of classes
- Like Factory pattern returned one of the several sub-classes, this returns such factory which later will return one of the sub-classes.

Abstract Factory Pattern



Singleton Design Pattern

- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.
- The singleton class must provide a global access point to get the instance of the class.
- Singleton pattern is used for logging, drivers objects, caching and thread pool.
- Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade etc.
- Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.



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

Builder Pattern

- The Builder pattern can be used to ease the construction of a complex object from simple objects.
- The Builder pattern also separates the construction of a complex object from its representation so that the same construction process can be used to create another composition of objects.
- Need of builder is more then that of factory pattern because we aren't returning objects which are simple descendents of a base display object, but totally different user interfaces made up of different combinations of display objects
- Separates object construction from its representation
- Builder pattern is useful when the construction of the object is very complex.
- The main objective is to separate the construction of objects and their representations. If we are able to separate the construction and representation, we can then get many representations from the same construction.
-

Prototype Pattern

- The prototype means making a clone.
- This implies cloning of an object to avoid creation. If the cost of creating a new object is large and creation is resource intensive, we clone the object.
- We use the interface Cloneable and call its method clone() to clone the object.

Structural Patterns

Adapter Pattern

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

When you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

- A class adapter
  - uses multiple inheritance (by extending one class and/or implementing one or more classes) to adapt one interface to another.
- An object adapter
  - relies on object aggregation.

Decorator design pattern

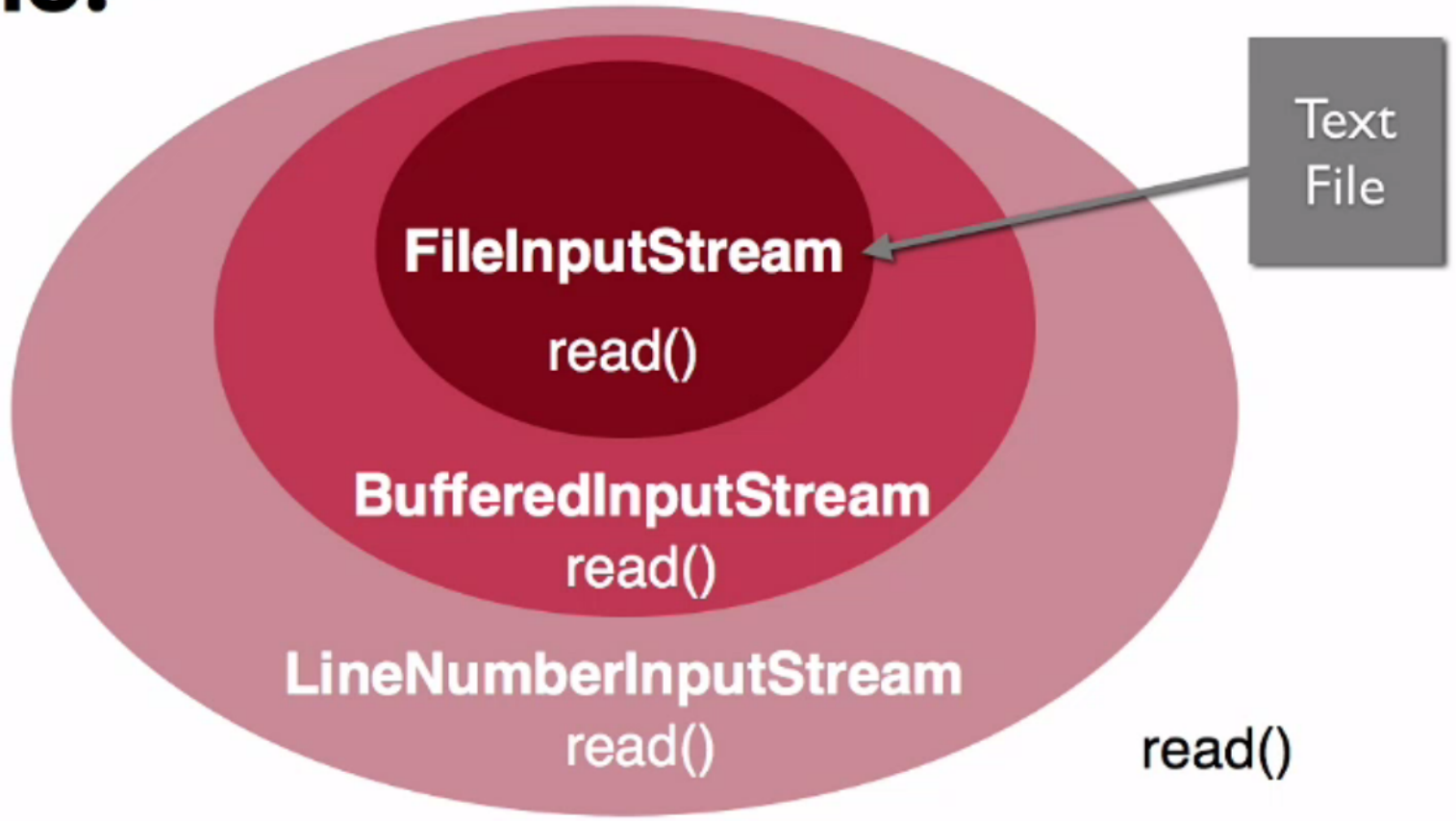
- Series of wrapper class that define functionality
- In the Decorator pattern, a decorator object is wrapped around the original object.
- The decorator must conform to the interface of the original object (the object being decorated).

Adding behaviour statically or dynamically

Extending functionality without effecting the behaviour of other objects.

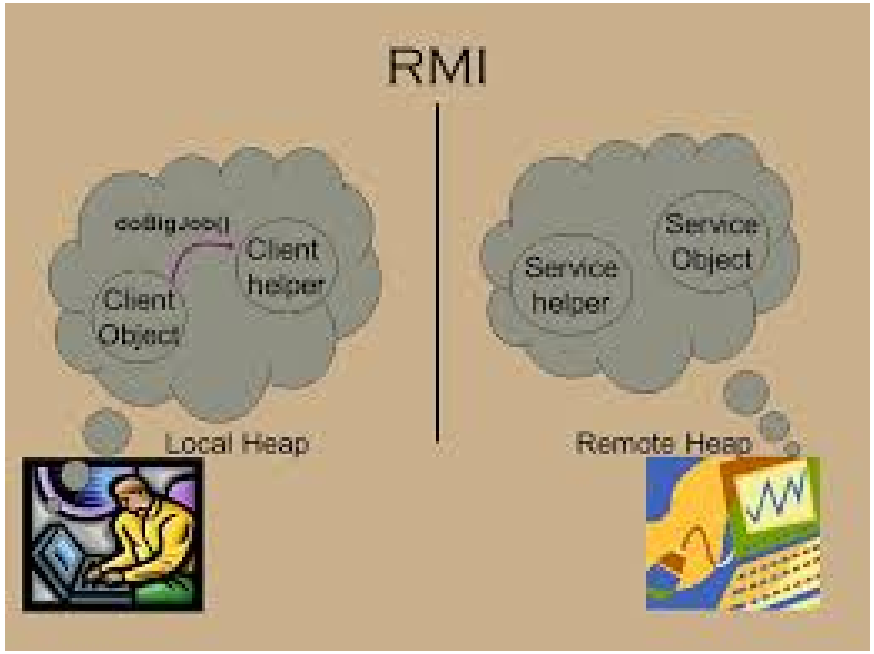
Adhering to Open for extension, closed for modification.

java.io.*


Text
File
FileInputStream
read()
BufferedInputStream
read()
LineNumberInputStream
read()
read()

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.



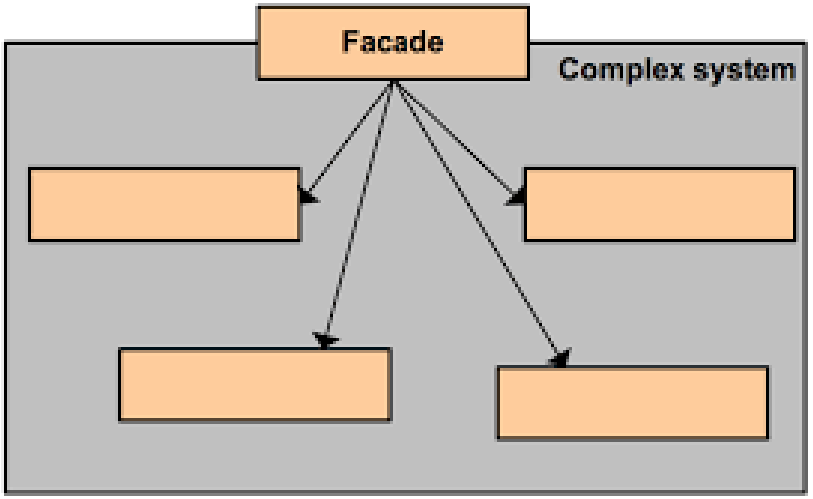
Facade pattern

The **facade pattern** (also spelled *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. A facade can

- make a software library easier to use, understand, and test, since the facade has convenient methods for common tasks,
- make the library more readable, for the same reason,
- reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system,
- wrap a poorly designed collection of APIs with a single well-designed API.

The Facade design pattern is often used when a system is very complex or difficult to understand because the system has a large number of interdependent classes or its source code is unavailable. This pattern hides the complexities of the larger system and provides a simpler interface to the client. It typically involves a single wrapper class that contains a set of members required by the client. These members access the system on behalf of the facade client and hide the implementation details.



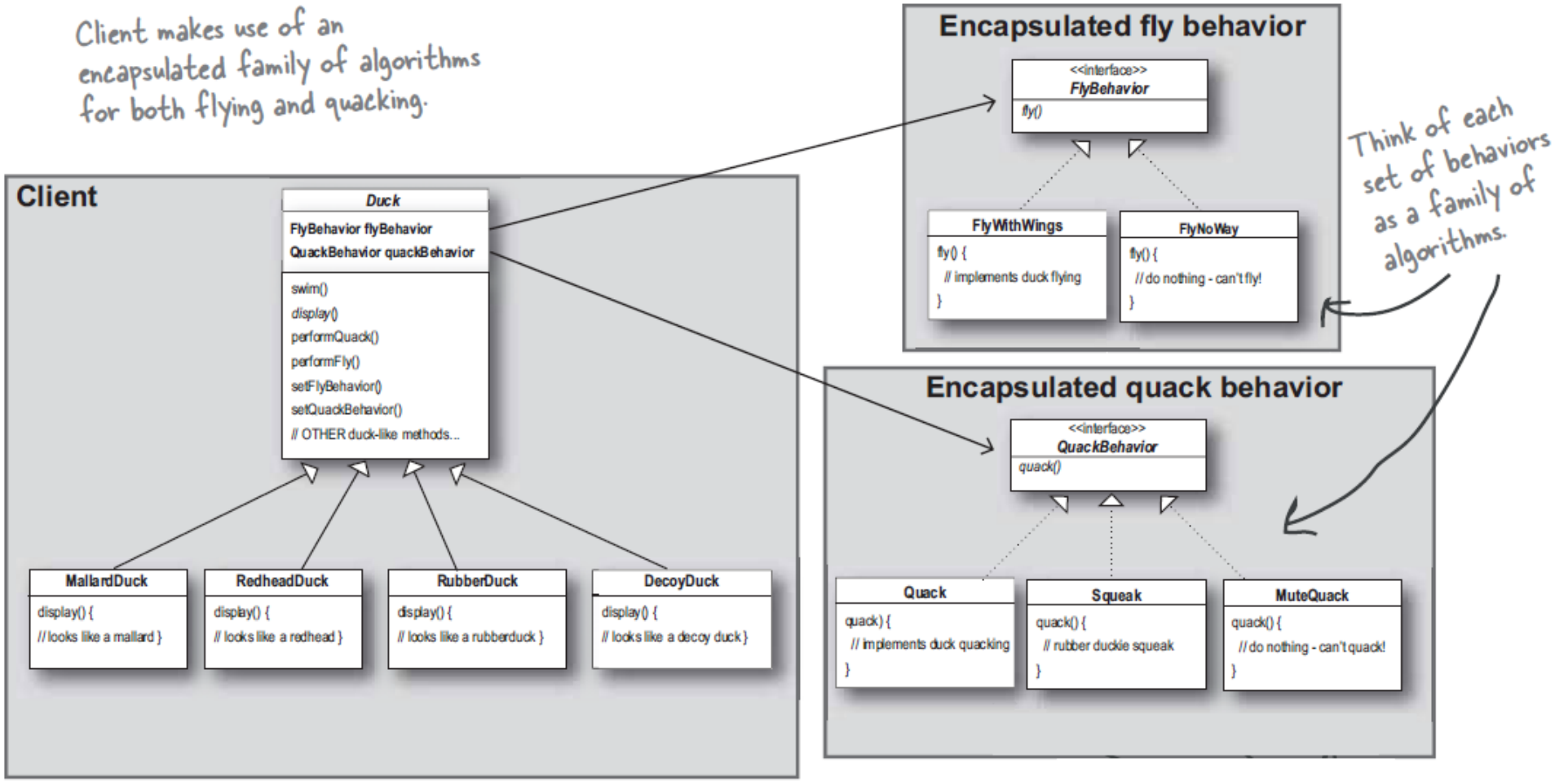
Behavioural Patterns

Strategy pattern /policy pattern

- The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable

- It is useful for situations where it is necessary to dynamically swap the algorithms used in an application.

- select algo at run time
- convert IS-A to HAS-A


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}

Iterator Design Pattern

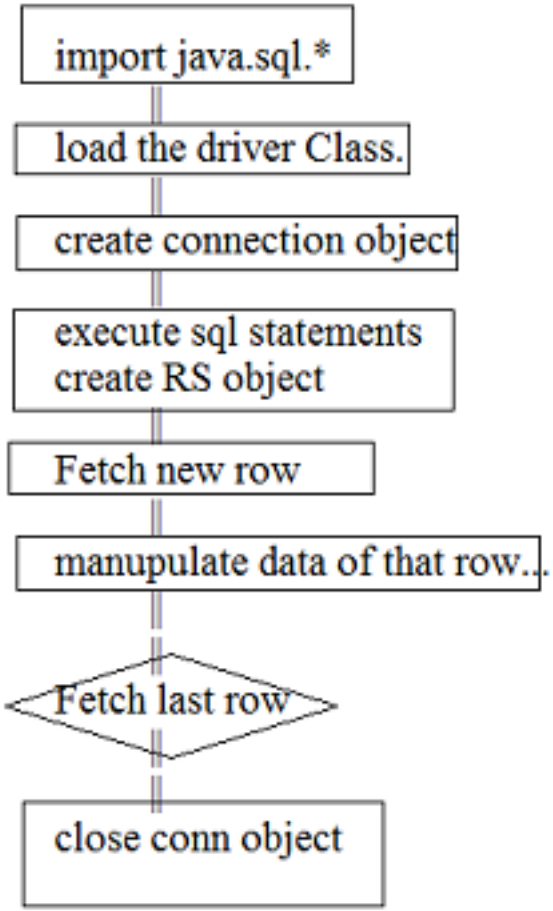
- The iterator pattern encapsulates iteration.
- The iterator pattern requires an interface called *Iterator*.
- The *Iterator* interface has two methods:
  - *hasNext()*
  - *next()*
- Iterators for different types of data structures are implemented from this interface.


DAY
9

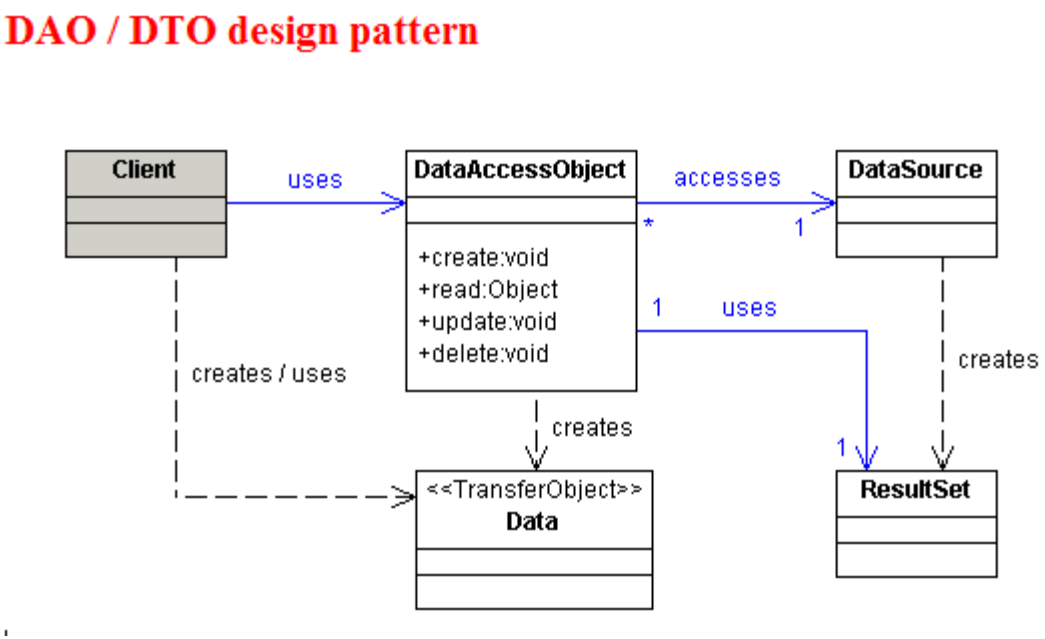
Day 9:Java Database Connectivity

- JDBC Fundamentals
- JDBC Drivers and architectures
- CURD operations using JDBC
- Autogenerated PK
- ResultSetMetaData, DataBaseMetaData
- Transaction management with JDBC

JDBC


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

DAO DTO design Pattern



rgupta.mtech@gmail.com