***Java Readmes***

***Day 1***

Refer: <slides>/PPTDay1

WHY Java?

1. Platform or architecture independent

(Write once run anywhere!)

2. Simple & robust

3. Secure

4. Automatic memory management.

5. Inherent Multi-threaded support

6. Object Oriented support -- Encapsulation, Inheritance & polymorphism, abstraction

7. Excellent I/O support

8. Inherent networking support for TCP/IP, UDP/IP programming & for URLs

9. Supports Functional programming

....

Development & Execution of java application.

JDK, JRE & JVM (refer to diagram)

Objective: Create a java appln to display welcome message on the console.

Naming conventions in Java

1. class, interfaces , Enum names- 1st letter of 1st word must start with upper case & then follow camel case notation.

eg : HelloWorld

2. data members/methods(functions) -- 1st must start with lower case & then follow camel case notation

eg : performanceIndex

calculateSalary

3. constants -- all uppercase.

eg : PI

4. Rules on Identifiers

1. Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore ( \_ ), cannot start with a number!

2. Can't use a Java keyword as an identifier.

3. Are Case sensitive

Legal Access specifiers for data members n methods

private : visible within the same class

default(package private) --no access modifier ---visible within same package

protected : accessible withing the same package & accessible to sub classess via inheritance

public : accessible from anywhere.

Legal class level access specifiers -

1. default(scope=current package only)

2. public (scope=accessible form any where)

How to compile ?

Change dir to <day1>/src

javac -d ..\bin HelloWorld.java

Objective --- Write Java application , to say hello to user. User name will be supplied as cmd line argument.

(eg : java SayHello Madhra

O/P Hello , Madhura !)

Objective : accept 2 nums as cmd line args , add them & display the result.

Java API -- java docs

java.lang => pkg name --default

Integer -- class

public static int parseInt(String s) throws NumberFormatException

(In case of un parseable integers.)

eg : int num1=Integer.parseInt(args[0]);

Reference for Java documentation

Java Docs.

Refer to java docs for any explanation of classes/methods..

Basic rules

1. Java compiler doesn't allow accessing of un initialized data members.

2. Files with no public classes(default scoped) can have a name that does not match with any of the classes in the file .

3. A file can have more than one non public class.

4. There can be only one public class per source code file.

5. If there is a public class in a file, the name of the file must match the name

of the public class.

For example, a class declared as public class Example {....}

must be in a source code file named Example.java.

6. Javac doesn't allow accessing of un-inited vars.

eg : int n;

sop(n);

Emp e;

sop(e);

String s="abc";

s++; //javac error -- pointer aritmetic not allowed in java

6. Package related rules later....

If the class is part of a package, the package statement must be the first line

in the source code file, before any import statements that may be present.

7. If there are import statements, they must go between the package statement

(if there is one) and the class declaration. If there isn't a package statement,

then the import statement(s) must be the first line(s) in the source code file.

If there are no package or import statements, the class declaration must be

the first line in the source code file.

8. import and package statements apply to all classes within a source code file.

In other words, there's no way to declare multiple classes in a file and have

them in different packages, or use different imports.

Java Data Types

refer to diagram from the slide

1. primitive data types

boolean,char,byte,short,long,float,double

What is unicode ?

refer to readme

2. reference types

class type , array type , interface type

Pointers vs java references

Pointer arithmatic is not allowed in java.

reference --- holds internal representation of address (equivalent to object pointer in c++)

Conversions regarding primitive types

Automatic conversions(widening ) ---Automatic promotions

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

char ---> int

long --->float ---is considered automatic type of conversion(since float data type can hold larger range of values than long data type)

Rules ---

src & destination - must be compatible, typically dest data type must be able to store larger magnitude of values than that of src data type.

1. Any arithmetic operation involving byte,short --- automatically promoted to --int

2. int & long ---> long

3. long & float ---> float

4. byte,short......& float & double----> double

Narrowing conversion --- forced conversion(type-casting)

eg ---

double ---> int

float --> long

double ---> float

Revise operators & control structures from : PPTDay1

Steps for attaching scanner, to accept inputs from User.(UI)

What is Scanner?

A class (java.util.Scanner) that represents text based parser(It has inherent small ~ 1K buffer)

It can parse text data from any source --Console input,Text file , socket, string

Steps

1. import java.util.\*; or import java.util.Scanner;

2. create instance of Scanner class

Scanner class constructor

public Scanner (InputStream in)

System.in --- standard input (stdin)

usage -- Scanner sc=new Scanner(System.in);

3. To check data type

public boolean hasNextInt(),

public boolean hasNextByte(),

public boolean hasNextLong()

...

4. To actually read n parse data

public int nextInt() throws InputMismatchException

public double nextDouble() throws InputMismatchException

public String next() throws InputMismatchException

public boolean nextBoolean() throws InputMismatchException

public String nextLine() throws InputMismatchException

5. Before terminating application close scanner.

public void close();

Solve : Write java application to accept int(emp id) , double(salary) , emp's first name , is permanent : boolean from Scanner & display the same using printf

***Day 2***

using printf

Today's topics

Continue with JVM Architecture

Enter OOPs -- encapsulation , data hiding

(class programming)

Packages , CLASSPATH

Revise

Java Data Types : which 2 categories

primitive types , referenece type

Types of conversions in primitive data types

widening(auto conversion / promotion) : performed by javac

eg : byte --> int , short --> double , long ---> float

narrowing conversion : performed by prog.(using type casting)

eg : double ---> float , int---> short

solve :

byte b1=10;

int data=b1;

float f=23.45;//javac err

double d1=f;

long l1=454654;

f=l1;

Answer this

What will data type of result of

1. byte & byte : int

2. byte & short : int

3. short & short : int

4. int & long : long

5. long & float : float

6. float & double : double

7. byte/short/int....& double : double

--------------------------------------------

JVM Architecture

Enter OOP

refer to "regarding class n object"

Encapsulation -- Class Programming (refer to

class,object,state,behaviour

Objective --

Create a class to represent 3D Box

class Box ---tight encapsulation(achieved by making all non static data members:instance vars private)

state -- width,height , depth --- double : private

paramterized constr --3 arg constr.

Instance Methods(=non static methods)

1. To return Box details in String form (dimensions of Box)

method declaration --- access specifier(private/default/protected/public) , ret type , name , args

method definition --body

2. To return computed volume of the box.

this --keyword in java

"this" represents => current object reference

Usages

1. To un-hide instance variable from local variable.

eg : this.width=width;

2. To invoke constructor of the same class , from different constructor.

eg : this(side,side,side);

Create another class(for the purpose of separation) : CreateBox --for UI.

Add psvmain(..){...}

Use scanner --to accept box dims from user, create box instance.

display its details & volume.

Regarding Garbage Collection

solve :

Box b1=new Box(1,2,3);

Box b2=b1;

Box b3=b1;

b1=b2=b3=null;//How many objects are marked GC ?

System.gc();//req --> accepted --which method GC thread will call auto? : finalize() .

-----------------

Pointers vs java references

Similarity -- Pointer & reference --hold an address to the object created on heap.

Difference -- To add robustness to the language ,

pointer arithmatic is not allowed in java.

reference --- holds internal representation of address --

eg :

int data =100;

data++;

Box b1=new Box(1,2,3);

b1++;

b1 += 10;

String s=new String("hello");

s += 10;

------------------

1. Create Cubes

2. Add a method to Box class to compare equality of 2 boxes & test it

3. Add a method to Box class to return a new Box with modified offset dims & test it with the tester.

Mem pictures

Garbage Collection

Enter packages

Day3

Regarding Packages

What is a package ?

Collection of functionally similar classes & interfaces.

Creating user defined packages

Need ?

1. To group functionally similar classes together.

2. Avoids name space collision (allows duplicate class names in different packages)

3. Finer control over access specifiers.

About Packages

1. Creation : package statement has to be placed as the 1st statement in Java source.

eg : package p1; => the classes will be part of package p1.

2.Package names are mapped to folder names.

eg : package p1; class A{....}

A.class must exist in folder p1.

3. For simplicity --- create folder p1 -- under <src> & compile from <src>

From <src>

javac -d ..\bin p1\A.java

-> javac will auto. create the sub-folder <p1> under the <bin> folder & place A.class within <p1>

NOTE : Its not mandatory to create java sources(.java) under package named folder. BUT its mandatory to store packged compiled classes(.class) under package named folders

Earlier half is just maintained as convenience(eg --- javac can then detect auto. dependencies & compile classes ).

3.5 How to launch / run packaged java classes?

cd <bin>

java FullyQualifiedClassName

java p1.A

4. To run the pkged classes from any folder : you must set Java specific environment variable : classpath

set classpath=g:\dac1\day2\bin;

classpath= Java only environment variable

Used mainly by JRE's classloader : to locate & load the classes.

Classloader will try to locate the classes from current folder, if not found --- will refer to classpath entries : to resolve & load Java classes.

What should be value of classpath ---Must be set to top of packged class hierarchy(eg : bin)

set classpath=d:\dac\day2\bin;.;(cmd line invocation)

OR better still

set it from environment variables.

-----------------------------------------------

Rules

1. If the class is part of a package, the package statement must be the first line in the source code file, before any import statements that may be present.

2. If there are import statements, they must go between the package statement

(if there is one) and the class declaration. If there isn't a package statement,

then the import statement(s) must be the first line(s) in the source code file.

If there are no package or import statements, the class declaration must be

the first line in the source code file.

3. import and package statements apply to all classes within a source code file.

In other words, there's no way to declare multiple classes in a file and have

them in different packages, or use different imports.

---------------------------

NOTE : Setting classpath on all platforms

Refer :

https://www.javacodestuffs.com/2020/09/how-to-set-classpath-in-java-windows.html#:~:text=%20How%20to%20set%20classpath%20in%20Java%20-,is%20one%20way%20to%20tell%20applications%2C...%20More%20

OR

<https://www.edureka.co/blog/set-Java-classpath/>

Day4

Inheritance

Regarding inheritance

In OOP, we often organize classes in hierarchy to avoid duplication and reduce redundancy. The classes in the lower hierarchy inherit all the variables (attributes) and methods (dynamic behaviors) from the higher hierarchies.

A class in the lower hierarchy is called a subclass (or derived, child, extended class). A class in the upper hierarchy is called a superclass (or base, parent class).

By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, redundancy can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. Re usability is maximum.

A subclass inherits all the member variables and methods from its superclasses (the immediate parent and all its ancestors). It can use the inherited methods and variables as they are. It may also override an inherited method by providing its own version, or hide an inherited variable by defining a variable of the same name.

Summary : Sub class IS-A super class , and something more (additional state + additional methods) and something modified(behaviour --- method overriding)

eg :

Person,Student,Faculty

Emp,Manager,SalesManager,HRManager,Worker,TempWorker

Shape, Circle,Rectangle,Cyllinder,Cuboid

BankAccount ,LoanAccount,HomeLoanAccount,VehicleLoanAccount,

Student,GradStudent,PostGradStudent

Fruit -- Apple -- FujiApple

A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors.

It is important to note that a subclass is not a "subset" of a superclass. In contrast, subclass is a "superset" of a superclass. It is because a subclass inherits all the variables and methods of the superclass; in addition, it extends the superclass by providing more variables and methods.

Inheritance --- generalization ----> specialization.

IS A Relationship.

Why -- code re usability.

super class ---base class

sub class --derived class

keyword --extends

Types of inheritance

1. single inheritance ---

class A{...} class B extends A{...}

Supported in java

2. multi level inhertance

class A{...} class B extends A{...} class C extends B{...}

Supported in java

3. multiple inhertiance --- NOT supported

class A extends B,C{...} -- compiler err

Why --For simplicity.

(Diamond problem)

We have two classes B and C inheriting from A. Assume that B and C are overriding an inherited method and they provide their own implementation. Now D inherits from both B and C doing multiple inheritance. D should inherit that overridden method. BUT which overridden method will be used? Will it be from B or C? Here we have an ambiguity.

Constructor invocations in inheritance hierarchy -- single & multi level.

eg -- Based on class A -- super class & B its sub class.

Further extend it by class C as a sub-class of B.

super keyword usage

1. To access super class's visible members(data members n methods)

eg : p1 : package

class A { void show(){sop("in A's show");}}

package p1 :

class B extends A {

//overriding form /sub class version

void show(){sop("in B's show");

super.show();

}

}

eg : B b1=new B();

b1.show();

2. To invoke immediate super class's matching constructor --- accessible only from sub class constructor.(super(...))

eg : Organize following in suitable class hierarchy(under "inheritance" package) : tight encapsulation

Person -- firstName,lastName

Student --firstName,lastName,grad year,course,fees,marks

Faculty -- firstName,lastName,yrs of experience , sme

Confirm invocation of constructors & super.

Regarding this & super

1. Only a constr can use this(...) or super(..)

2. Has to be 1st statement in the constructor

3. Any constructor can never have both ie. this() & super()

4. super & this (w/o brackets) are used to access (visible) members of super class or the same class.

eg :

1.Simple example to understand inheritance n polymorphism

1.1 Fruit : name

Add a parametrized constr , to accept name of the fruit.

Add taste() method to display its taste.

eg : public void taste() : "no specific taste"

1.2 Apple : extends Fruit

parametrized constr ---super(name);

override : taste

method definition : sweet n sour in taste

1.3 Similarly : add Orange n Mango

parametrized constr ---super(name);

Add taste() method to display its taste.

Orange : Sour in taste

Mango : sweet in taste

1.5 Write a simple tester : to understand upcasting n run time polymorphism.

2. Another example

Write a Tester to create basket of fruits.

(populate basket based on user choice)

1. Fruit,Apple,Orange,Cherry

Add taste() method to display its taste.

2. Create FruitUtils class.

Add static method , addFruit to add a fruit to the Fruit Basket.

3. Write a Tester to create basket of fruits.

(populate basket based upon user's choice)

Menu

1. Add Apple

2. Add Orange

3. Add Cherry

4. Display taste of all fruits in the basket.

5. Exit : terminate the application.

Menu

1. Add Apple

2. Add Orange

3. Add Mango

4. Display taste of all fruits in the basket (for-each)

5 : Exit : terminate the application.

2. Create FruitUtils class.(later)

Add static method , addFruit to add a fruit to the Fruit Basket.

Example 2 (Lab work)

1. Shape -- x,y

Method --public double area()

public String toString()

2. Circle -- x,y,radius

Method --public double area()

public String toString()

3. Rectangle -- x,y,w,h

Method --public double area()

public String toString()

4. Square-- x,y,side

Method --public double area()

public String toString()

5. Create a ShapeFactory class

Add a method(generateShape) to return randomly generated shape.

6. Create a Tester . Invoke ShapeFactory's generateShape() method , in a for-loop

to display details & area of each shape.

***POLYMORPHISM***

Polymorphism ---one functionality --multiple (changing) forms

1. static -- compile time --early binding ---resolved by javac.

Achieved via method overloading

rules -- can be in same class or in sub classes.

same method name

signature -- different (number/type/both)

ret type --- ignored by compiler.

eg --- void test(int i,int j){...}

void test(int i) {..}

void test(double i){..}

void test(int i,double j,boolean flag){..}

int test(int a,int b){...}

RULE -- when javac doesn't find exact match --tries to resolve it by the closest arg type(just wider than the specified arg)

solve --- EasyOver.java

(More interesting examples after boxing & var-args)

2. Dynamic polymorphism (run time polymorphism) --- late binding --- dynamic method dispatch ---resolved by JRE.

Dynamic method dispatch -- which form of method to send for execution ---This decision can't be taken by javac --- BUT taken by JRE

Achieved via -- method overriding

Method Overriding --- Means of achieving run-time polymorphism

NO "virtual" keyword in java.

All java methods can be overridden : if they are not marked as private,static,final

Super-class form of method - --- overridden method

sub-class form --- overriding form of the method

Rules : to be followed by overriding method in a sub-class

1. same method name, same signature, ret type must be same or its sub-type(co-variance)

eg of co-variance

class A {

A getInstance()

{

return new A();

}

}

class B extends A

{

B getInstance()

{

return new B();

}

}

2. scope---must be same or wider.

3. Will be discussed in exeception handling.

Cannot add in its throws clause any new or broader checked exceptions.

BUT can add any new unchecked excs.

Can add any subset or sub-class of checked excs.

class A

{

void show() throws IOExc

{...}

}

class B extends A

{

void show() throws Exc

{...}

}

Can't add super class of the checked excs.

example of run time polymorphism -- Car & its sub classes.

From JDK 1.5 onwards : Annotations are available --- metadata meant for Compiler or JRE.(Java tools)

Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in java are used to provide additional information, so it is an alternative option for XML.

eg @Override,@Deprecated,@SuppressWarnings,@FunctionalInterface

@Override --

Annotation meant for javac.

Method level annotation

Optional BUT recommended.

eg : Fruit <----- Orange

public class Orange extends Fruit {

@Override

public void taste() {....}

}

While overriding the method in a sub class -- if you want to inform the compiler that : following is the overriding form of the method use :

@Override

method declaration {...}

Run time polymorphism or Dynamic method dispatch in detail

Super -class ref. can directly refer to sub-class object(direct=w/o type casting) as its the example of up-casting(similar to widening auto. conversion) .

When such a super class ref is used to invoke the overriding method: which form of the method to send for execution : this decision is taken by JRE & not by compiler. In such case --- overriding form of the method(sub-class version) will be dispatched for exec.

Super -class ref. can directly refer to sub-class inst BUT it can only access the members declared in super-class -- directly.

eg : A ref=new B(); ref.show() ---> this will invoke the sub-class: overriding form of the show () method

----------------------------------------------

Applying inheritance & polymorphism

java.lang.Object --- Universal super class of all java classes including arrays.

Object class method

public String toString() ---Rets string representation of object.

Returns --- Fully qualified class Name @ hash code

hash code --internal memory representation.(hash code is mainly used in hashing based data structures -- will be done in Collection framework)

Why override toString?

To replace hash code version by actual details of any object.

Objective -- Use it in sub classes. (override toString to display Account or Point2D or Emp details or Student / Faculty )

--------------------------

Object class method

public boolean equals(Object o)

Returns true --- If 'this' (invoker ref) & o ---refers to the same object(i.e reference equality) i.e this==o , otherwise returns false.

Need of overriding equals method ?

To replace reference equality by content identity equality , based upon prim key criteria.

eg : In Car scenario

(Primary key -- int registration no)

Objective : use it for understanding downcasting n instanceof keyword

------------------------------

instanceof -- keyword in java --used for testing run time type information.

refer : regarding instanceof

Solve

Fruit f=new Fruit();

f.taste();

f.pulp();

((Mango)f).pulp();

f=new Orange();

f.taste();

((Mango)f).pulp();

if(f instanceof Mango)

((Mango)f).pulp();

else

sop("Invalid fruit....");

if(f instanceof Object)

((Mango)f).pulp();

else

sop("Invalid fruit....");

--------------------------------

abstract : keyword in Java

1. abstract methods ---methods only with declaration & no definition

eg : In BoundedShape class , add abstract method declaration fro area computation (since it can't supply implementation details)

public abstract double area();//completely equivalent to CPP 's pure virtual function

2. private abstract method ? javac error

3. Any time a class has one or multilple abstract methods ---- class must be declared as abstract class.

eg. public abstract class BoundedShape{....}

4. Abstract classes can't be instantiated. BUT can create the reference of abstract class type to refer to concrete sub-class instances.

eg : BoundedShape shape=new BoundedShape(....);//javac err : Invalid RHS

OR

BoundedShape shape=new Circle(.....);// no err

sop(shape.area());//JVM : area() will be invoked dynamically : Circle's object

5. Abstract classes CAN HAVE concrete(non-abstract) methods.

6. Abstract classes MUST provide constructor/s to init its own private data members.(for creating concrete sub class instance)

eg : BoundedShape : private int x,y;

Rectangle , Triangle .....

7. Can a class be declared as abstract & final ? NO (javac error)

8. Can an abstract class be crerated with 100% concrete functionality?(no abstract methods)

Yes

eg --- Event adapter classes / HttpServlet

Use "abstract" keyword in Emp , Mgr ,Worker hierarchy & test it

final -- keyword in java

Usages

1 final data member(primitive types) - constant.

eg -- public final int data=123;

2. final methods ---can't be overridden.

usage eg public final void show{.....}

eg -- Object class -- wait , notify ,notifyAll

3. final class --- can't be sub-classed(or extended) -- i.e stopping inheritance hierarchy.

eg -- String ,StringBuffer,StringBuilder

eg : public class MyString extends String {...} //javac err

4. final reference -- references can't be re-assigned.

eg --final Emp e=new Mgr(.......);//up casting

e=new Worker(.....);//compiler err

--------------------

Special note on protected

Protected members act as default scope within the same package.

BUT outside pkg -- a sub-class can access it through inheritance(i.e just inherits it directly) & CAN'T be accessed by creating super class instance.

Private Members

Do subclasses inherit private data members from it's superclass?

NO !

Explanation :

As per the java language specification :

Members of a class that are declared private are not inherited by subclasses of that class. Only members of a class that are declared protected or public are inherited by subclasses declared in a package other than the one in which the class is declared.

BUT what we mean here by inheritance is , are private members accessible in a subclass ?

That answer is NO

BUT , sub class instance DOES CONTAIN private fields of their superclasses .

eg : Person has data members : private firstName , lastName

Student extends Person

It has ADDITIONAL data members :private gradYear,course,fees,marks

Answer this !

Can you access firstName & lastName from Student class ? NO

Student IS-A Person

So when you create an instance of a Student : firstName n lastName will be present in Student object , mem allocated in heap.

So how many slots will you show in Student object ?

CP + 6 slots .

*Static*

static --- keyword in java

Usages

1. static data members --- Memory allocated only once at the class loading time --- not saved on object heap --- but in special memory area -- method area (meta space) . -- shared across all objects of the same class.

Initialized to their default values(eg --double --0.0,char -0, boolean -false,ref -null)

How to refer ? -- className.memberName

eg -- public static int idCounter;

2. static methods --- Can be accessed w/o instantiation. (ClassName.methodName(....))

Can't access 'this' or 'super' from within static method.

Rules -- 1. Can static methods access other static members directly(w/o instance) -- YES

2. Can static methods access other non-static members directly(w/o instance) -- NO

eg : class A

{

private int i;

private static int j;

public static void show()

{

sop(i);//javac err

sop(j);//no err

}

}

3. Can non-static methods access other static members directly(w/o instance) -- YES

eg :

In Test class

void test1() {test2();}//no error

OR

static void test2(){test1();//javac error}

3. static import --- Can directly use all static members from the specified class.

eg --

//can access directly , ALL static members of the System class

import static java.lang.System.\*;

import static java.lang.Math.sqrt;

import java.util.Scanner;

main(...)

{

out.println(.....);

Scanner sc=new Scanner(in);

sqrt(12.34);

gc();

exit(0);

}

4. static initializer block

syntax --

static {

// block gets called only once at the class loading time , by JVM's classloader

// usage --1. to init all static data members

//& can add functionality -which HAS to be called precisely once.

Use case : singleton pattern , J2EE for loading hibernate/spring... frmwork.

}

They appear -- within class definition & can access only static members directly.(w/o instance)

A class can have multiple static init blocks(legal BUT not recommended)

Regarding non-static initilizer blocks(instance initilaizer block)

syntax

{

//will be called per instantiation --- before matching constructor

//Better alternative --- parameterized constructor.

}

5. static nested classes ---

eg --

class Outer {

// static & non-static members

static class Nested

{

//can access ONLY static members of the outer class DIRECTLY(w/o inst)

}

}

***Upcasting***

Upcasting

The most important aspect of inheritance is the relationship expressed between the new class and the base class. This relationship can be summarized by saying,

The new class "IS A" type of the existing class.

eg : Student is of Person type or Faculty is of Person type.

This description is not just a fancy way of explaining inheritance—it’s supported directly by the language.

Meaning :

Can we say ?

Person p=new Student(....);//YES --upcasting

sop(p);//dynamic method dispatch

As another example, consider a base class called Fruit that represents any fruit, and a derived class called Mango. Because inheritance means that all of the methods in the base class are also available in the derived class, any message you can send to the base class can also be sent to the derived class. If the Fruit class has a taste( ) method, so will Mango.

This means we can accurately say that a Mango object is also a type of Fruit.

Day5

Abstraction

Abstraction is the property with which only the essential details are displayed to the user.

The internal details or the non-essentials details are not displayed to the user.

(Hiding complexities or hiding the implementation details from end user)

Eg: An ATM is considered as just money rendering machine rather than its internal complex details

Consider a real-life example of a person walking to an ATM She only knows how to withdraw / deposit money. But as the end user , she does not really need to know about how ATM connects with the underlying bank to inform about this transaction ...

This is what abstraction is.

In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

When to use abstract classes and abstract methods ?

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.(i.e only provide declaration)

Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

eg : BoundedShape & it's method area

Abstract classes and Abstract methods :

refer to readme

Instance of

instanceof -- keyword in java --used for testing run time type information.(RTTI)

It is used to test whether the object is an instance of the specified type (class or subclass or interface).

Meaning

In "a instanceof B", the expression returns true if the reference to which a points is an instance of class B, a subclass of B (directly or indirectly), or a class that implements the B interface (directly or indirectly).

The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false.

For null --instanceof returns false.

For sub-class object --instanceof super class -- rets true

For super-class object --instanceof sub class -- rets false

eg ---Object <----Emp <---Mgr <---SalesMgr

Object <---- Emp <--- Worker

What will be o/p ?

Emp e =new Mgr(...);//up casting

e instanceof Mgr -true

e instanceof Emp --true

e instanceof Object --true

e instanceof SalesMgr -- false

e instanceof Worker -- false

e=null;

e instanceof Emp/Mgr/SalesMgr/Worker/Object --- false

***Can’t OverRide Static methods, why?***

Why static methods can't be overridden in java ?

Method overriding is a way to achieve dynamic method dispatch (i.e run time polymorphism)

Meaning which behavior to choose or which method to choose for execution --this decision is taken at the run time depending upon type of the object by the JVM(late binding). Since it depends upon the type of the object , for static methods this concept is not applicable. (since they are not associated with any object)

Overriding depends on having an instance of a class. The point of polymorphism is that you can subclass a class and the objects implementing those subclasses will have different behaviors for the same methods defined in the superclass (and overridden in the subclasses). A static method is not associated with any instance of a class so the concept is not applicable.

**Day 6**

**INTERFACE**

Interface in Java

What is interface ?

An interface in java is a blueprint of a class. It has public static final data members and public n abstract methods only.

The interface in java is a mechanism to achieve fully abstraction. There can be only abstract methods in the java interface (not method body)(true till JDK 1.7) . It is used to achieve full abstraction and multiple inheritance in Java.

Java Interface also represents IS-A relationship.

It cannot be instantiated just like abstract class.

Why java interfaces?

1. It is used to achieve full abstraction.

2. By interface, we can support the functionality of multiple inheritance.

3. It can be used to achieve loose coupling.

(Interfaces allow complete separation between WHAT(specification or a contract) is to be done Vs HOW (implementation details) it's to be done

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

---------------------------

syntax of interface

default(no modifier)/public interface NameOfInterface extends comma separated list of super interfaces

{

//data members --- public static final : added implicitly by javac

int DATA=100;

//methods -- public abstract : added implicitly by javac

double calc(double d1,double d2);

}

Implementing class syntax

default(no modifier)/public class NameOfClass extends SuperCls implements comma separated list of interfaces {

//Mandatory for implementation class to be non-abstract(concrete): MUST define/implement all abstract methods from all i/fs.

}

eg : public class Circle extends Shape implements Computable,Runnable {...}

1. Relationship between classes and interfaces

A class inherits from another class(extends), an interface extends another interfaces(extends) but a class implements an interface.

2. Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

eg :

Multiple inheritance in java

interface Printable{

void print();

}

interface Showable{

void show();

}

class A implements Printable,Showable{

public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}

public static void main(String args[]){

A obj = new A();

obj.print();

obj.show();

}

}

Question

Multiple inheritance is not supported through class in java but it is possible by interface, why?

Multiple inheritance is not supported in case of class, sincee it can create an ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

For example:

interface Printable{

void print();

}

interface Showable{

void print();

}

class TestTnterface1 implements Printable,Showable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

TestTnterface1 obj = new TestTnterface1();

obj.print();

}

}

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

interface Printable{

void print();

}

interface Showable extends Printable{

void show();

}

class Testinterface2 implements Showable{

public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}

public static void main(String args[]){

Testinterface2 obj = new Testinterface2();

obj.print();

obj.show();

}

}

Q) What is marker or tagged interface?

An interface that has no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM(Run time marker) so that JVM may perform some useful operation.

//How Serializable interface is written?

public interface Serializable{

}

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface.

eg :

interface printable{

void print();

interface MessagePrintable{

void msg();

}

}

Q . What is a functional i/f

An interface containing sing abstract methods (SAM)

eg : Comparator , Runnable , Consumer...

***Abstract Classes vs Interface***

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface.

Abstract class Vs Interface

1) Abstract class can have abstract and non-abstract methods. Interface can have only abstract methods.

2) Abstract class doesn't support multiple inheritance. Interface supports multiple inheritance.

3) Abstract class can have final, non-final, static and non-static variables. Interface has onlypublic,static and final variables.

4) Abstract class can have static methods, main method and constructor. Interface can't have static methods, main method or constructor.

5) Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.

6) The abstract keyword is used to declare abstract class. The interface keyword is used to declare interface.

7) Example:

public abstract class Shape{

public abstract void draw();

} Example:

public interface Drawable{

void draw();

}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Abstract Class vs. Interface

Java provides and supports the creation of abstract classes and interfaces. Both implementations share some common features, but they differ in the following features:

1. All methods in an interface are implicitly abstract. On the other hand, an abstract class may contain both abstract and non-abstract methods.

2.A class may implement a number of Interfaces, but can extend only one abstract class.

3.

In order for a class to implement an interface, it must implement all its declared methods. However, a class may not implement all declared methods of an abstract class. Though, in this case, the sub-class must also be declared as abstract.

Abstract classes can implement interfaces without even providing the implementation of interface methods.

4.

Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.

5.

Members of a Java interface are public by default. A member of an abstract class can either be private, protected or public.

6.

An interface is absolutely abstract and cannot be instantiated, doesn't support a constructor. An abstract class also cannot be instantiated BUT can contain a constructor to be used while creating concrete(non abstract) sub class instance.

DAY 7

***EXCEPTION HANDLING***

Exception Handling

Regarding Exception Handling in java.....

Any run time err occurs(eg file not found,accessing out of array size,accessing func from null ref, divide by 0)

---JRE(main thrd) --- creates matching exc class instance(java.io.FileNotFoundException,java.lang.ArrayOutOfBoundsExc,NullPointerExc,ArithmeticExc)

--- JRE checks -- if prog has proivided exc handling code ?

--- NO -- JRE aborts java code(by supplying def handler) & prints details --F.Q exc class name,reason behind failure & location details(err stack trace

--- YES (try---catch) JRE execs exc handling block & continues with the rest of the code.

syntax(key words) --- try,catch,finally,throw,throws

Inheritance hierarchy of exc classes

unchecked vs checked excs.

Creating custom excs

JDK 1.7 syntax --- try-with-resources(in I/O or device prog)

Checked & Unchked exception are detected or occur only in run-time.

JRE/JVM DOES NOT distinguish between them

Compiler(javac) differentiates bet them

Javac forces handling of the checked exc. upon the prog.(Handling by supplying matching try-catch block or including it in the throws clause.)

Legal syntax

1. try {...} catch (exc1 e){...}

2. try {...} catch (exc1 e){...} catch (exc2 e) {..} ....

3. try {...} catch (NPE e){} catch (AE e) {}catch(Exception e){catch-all}

3.5 3. try {...} catch (AE e){...} catch (NPE | AOB e) {...}catch(Exception e){catch-all}

4. throws syntax ---

method declaration throws comma separated list of exc classes.

eg : Integer class API

public static int parseInt(String s) throws NumberFormatException

Thread class API

public static void sleep(long ms) throws InterruptedException

FileReader API

public FileReader(String fileName) throws FileNotFoundException

throws --- keyword meant for javac

Meaning -- Method MAY raise specified exc.

Current method is NOT handling it , BUT its caller should handle.

Mandatory--- only in case of un handled(no try-catch) chked excs(not extended from RuntimeException).

Use case --used in delegating the exception to caller.

4.5 Throwable class API

1. public String toString() -- rets Name of exception class & reason.(detailed err mesg)

2. public String getMessage() -- rets error mesg of exception

3. public void printStackTrace() --- Displays name of exc class, reason, location dtls.

5. finally --- keyword in exc handling

finally -- block -- finally block ALWAYS survives(except System.exit(0) i.e terminating JVM)

i.e in the presence or absence of excs.

5.1 try{...} catch (Exception e){....} finally {....}

5.2 try{...} catch (NullPointerException e){....} finally {....}

5.3 try {...} finally {....}

try-with-resources

From Java SE 7 onwards --- Java has introduced java.lang.AutoCloseable -- i/f

It represents --- resources that must be closed -- when no longer required.

Autocloesable i/f method

public void close() throws Exception-- closing resources.

Java I/O classes(eg : BufferedReader,PrintWriter.....),Scanner -- have already implemented this i/f -- to automatically close resource when no longer required.

syntax of try-with-resources

try (//can open one or multiple AutoCloseable resources)

{ ......

} catch(Exception e)

{

}

eg :

try(Scanner sc=new Scanner(System.in);

FileReader fr=new FR(....))

{

..........

} catch -all

Creating Custom Exc(User defined exception or application exc)

Need :

1. Validations : In case of validation failures : Prog will have to throw custom exc class instance

2. B.L failures (eg : funds transfer : insufficient finds) : Prog will have to throw custom exc class instance

1. Create a pkged public class which extends Throwable(not reco but legal)/Exception(recommended)/Error(not reco but legal)/RuntimeExc(not reco but legal)

eg : public class MyException extends Exception{

public MyException(String mesg)

{

super(mesg);

}

}

public class MyException2 extends RunTimeException{....}

2.CustExc(String msg) : overload the constr : to invoke the super-class constr.

of the form

Exception (String msg)

OR

CustExc(String msg,Throwable rootCause)

public Exception(String message,Throwable cause)

Objective :

Check the speed of vehicle on a freeway

Accept the speed using Scanner : can be speed too low(exc) or too high(exc) or in range

keyword -- throw --for throwing exception.

JVM uses it to throw built-in exceptions(eg : NullPointerExc , IOException etc) & prog uses it throw custom exception(user defined excs) in case of B.L or validation failures.

syntax :

throw Throwable instance;

eg :

throw new NullPointerExc();// no javac err

throw new InterruptedExc();// no javac err

throw new Throwable("abc");// no javac err

throw new Account(...);//javac err (provided it doesn't extend from Throwable hierarchy)

throw new AccountOverdrawnException("funds too low...");//proper usage

**Day9**

String API

String class API

Important String class constructors

1.String(byte[] bytes) --- byte[] ----> String converter

2.String(char[] chars) --- char[] ---> String converter

3.String (byte[] bytes,int offset,int len) ---byte[] ----> String converter from the specified offset , specified len no of bytes will be converted.

eg . String s=new String(bytes,3,4); --- String will contain bytes[3]----bytes[6]

4. String(char[] ch,int offset,int len)

5. String(String s)

String class methods --- to go through

charAt,compareTo,contains,copyValueOf,format,valueOf,getBytes,toCharArray,toLowerCase,indexOf,lastIndexOf,split,replace,startsWith,endsWith,length,intern

1.

boolean equals(Object o) ---- ret true iff 2 strings are having same contents (case sensitive)

About equals()

super class def. --- java.lang.Object

public boolean equals(Object o)

Rets true iff both refs(this & o) are equal i.e referring to the same object.

Sub-class developers MUST override equals for content-wise(depending on Object's state) comparison.

2. concat,charAt,indexOf,lastIndexOf,toUpperCase,toLowerCase,format,split

printf & Formatter class

Refer to java.util.Formatter class for formatting conversion details.

Imp ---

Formatting details

%c -- character

%b -- boolean

%h -- hex value of hashcode of obj ref.

%s -- string

%d -- int

%f,%g -- float/double

%x -- hex value

%n -- line separator

%tD -- Date

%tT -- Time

%tc -- Time stamp(date & Time)

%td-%1$tb-%1$tY -- can be applied to GC or Date.

**Date/Time Handling in Java**

API

1. java.util.Date--- represents system date , till the msec precision

Constructor

1.Date() --- creates Date class instance representing system date, current date.(till ms precision)

2.Date(long msec) --- creates Date class instance representing date for msec elapsed after epoch(=1st Jan 1970)

eg : Date d1=new Date(100);

Methods --toString,before,after,equals,compareTo

---------------------

For parsing(string-->Date) & formatting (Date --> String)

1. Create an instance of java.text.SimpleDateFormat (extends DateFormat)

Constr : SimpleDateFormat(String pattern)

pre defined pattern

y --year

MM -- month in digit(1-12)

MMM -- month in abbrevation(Jan,Feb...)

MMMM ---complete month name

d -day

h- Hour

m --minute

s -- second

eg : SimpleDateFormat sdf=new SimpleDateFormat("dd-MM-yyyy , hh:mm:ss");

OR

SimpleDateFormat sdf=new SimpleDateFormat("dd/MM/yyyy");

2. Parsing (use inherited API) string ----> Date

public Date parse(String s) throws ParseException

3. Formatting

public String format(Date d)

2. java.util.GregorianCalendar

month range --- 0-11

GregorianCalendar(int yr,int mon,int date);

GregorianCalendar(int yr,int mon,int date,int hr,int min,int sec);

2.5 How to find out current year ?

GregorianCalendar class API (inherited from Calendar class)

public int get(String fieldName)

eg : gc.get(Calendar.YEAR);

3. Date/Time formatting via printf

%tc -- for complete timestamp(date & time)

%tD -- for date

%tT -- time

Arguments --- Date, GregorianCalendar

static import syntax ---

eg -- import static java.util.Calendar.\*;

or import static java.lang.System.\*;

in such src - u can access directly static members of Calendar class or from 2nd statement u can directly use out.println("testing static imports!");

var-args ...

variable args syntax.--- Must be last arg in the method args.

Can use primitive type as well as ref types.

Legal ---

void doStuff(int... x) {

....

} // expects from 0 to many ints

Usage : ref.doStuff();//no args

int[] ints={1,2,3,4};

ref.doStuff(ints);//array

ref.doStuff(20,34,56);/ comma separated list of args

System.out.printf("%n");//legal

System.out.printf(1234);//javac error

// as parameters

void doStuff2(char c, int... x) {..... } // expects first a char,

// then 0 to many ints

class Test {

void doStuff3(Animal... animals) {

for(Animal a : animals)

sop(a.getName());

} // 0 to many Animals

}

Test ref=new Test();

invocations ---

ref.doStuff3();//no javac err

Animal animals[]={new Cat(),new Dog(),new Horse()};//4 objs

ref.doStuff3(animals);//no javac err

Animal a1=new Horse();

Animal a2=new Cat();

Animal a3=new Dog();

ref.doStuff3(a1,a2,a3);

Illegal: javac error

void doStuff4(int x...) {....} // bad syntax : javac err

void doStuff5(int... x, char... y) {...} // too many var-args

void doStuff6(String... s, byte b) { } // var-arg must be last

***Day10***

***ASSOCIATION***

Regarding Association

Association is relationship between two separate classes , using object references.

Represents HAS-A

Why : Code reusability

Association can be one-to-one, one-to-many, many-to-one, many-to-many.

Composition and Aggregation are the two forms of association.

eg : refer to association-aggregation-composition.png

Association :

Owner HAS-A Pet -- Owner feeds a Pet & Pet plays with Owner.

Aggregation implies a HAS-A relationship where the child can exist independently of the parent.

eg : Class & Student / Bank HAS-A Customer

class Bank

{

private String name;

private String ifsc;

private String address;

//one to many

private Customer[] customers;

}

class Customer {...

+

//many to one

private Bank myBank;

}

Composition (Part Of or Belongs To)

Pet HAS-A Tail

It implies a relationship where the child cannot exist independent of the parent.

eg : Human HAS-A Lungs / Car HAS-A Engine / Person HAS-A Address

(when parent is deleted , typically child cant't exist on its own)

eg :

class Person

{

private String firstName,lastName;

private Date dob;

private String uid;

private Address adr;

private class Address

{

private String street,city,state,country;

....

}

//setter / method

}

Aggregation is a weaker form of HAS-A relationship than Composition

ENUMS

What is enum in java ?

Enumerations (in general) are generally a set of related constants.

They have been in other programming languages like C++ from beginning. BUT more powerful in Java.

Supported in Java since JDK 1.5 release.

Enumeration in java is supported by keyword enum. enums are a special type of class that always extends java.lang.Enum.

It's a combination of class & interface features.

Why ?

1. Helps to define constants.

2. Adds type safety to constants.

eg interface MovieConstants

{

int AGE\_MINOR=16;

int AGE\_MIN = 10;

int AGE\_MAX=70;

int TKT\_COST\_SILVER =100;

int TKT\_COST\_GOLD =200;

int TKT\_COST\_PLATINUM =300;

}

If by programer's mistake application uses TKT\_COST to compare ages of user , what will happen ?

Both being int type neither javac or jvm can realise err , but you will get wrong results.

It should not be allowed --as they represent different types ---AGE type & TKT\_COST type.

3. You can't iterate over all constant values from i/f but with enums you can.

4 . Consider this

eg interface Menu

{

String SOUP="Tomato soup";

String DOSA="Mysore Dosa";

String RICE="Fried rice";

}

Can you assign any price along with menu? ---Not easily !

But with enums you can.

----------------------------

A simple usage will look like this:

public/default enum Direction {

EAST,

WEST,

NORTH,

SOUTH //optionally can end with ";"

}

Here EAST, WEST, NORTH and SOUTH are implicitely of type

public final static Direction EAST=new Direction("EAST",0) ---super("EAST",0);

public final static Direction WEST=new Direction("WEST",1) ---super("WEST",1);

Super class of all enums

public abstract class Enum<E extends Enum<E>>

extends Object

implements Comparable<E>, Serializable

ie. they are comparable and serializable implicitly.

All enum types in java are singleton by default.

So, you can compare enum types using ‘==’ operator also.

Since enums extends java.lang.Enum, so they can not extend any other class because java does not support multiple inheritance . But, enums can implement any number of interfaces.

enum can be declared within a class or separately.

eg of enum within a class

When declared inside a class, enums are always static by default

eg public class TestOuter

{

enum Direction

{

EAST,

WEST,

NORTH,

SOUTH

}

}

To access a direction -- use TestOuter.Direction.NORTH.

Constructors of enum

By default, you don't have to supply constructor definition.

Javac implicitely calls super class constructor , Enum(String name,int ordinal)

Important Methods of Enum (implicitly added by javac)

1. public static Enum[] values() --rets array of enum type of refs.--pointing to singleton objs

2. public static Enum valueOf(String name) throws IllegalArgumentException -- string to enum type converter

values & valueOf methods generated by compiler --so not part of javadocs.

If you pass a different name (eg -- ABC) to valueOf ---throws IllegalArgumentException

Inherited from Suerpclass Enum

String name() --rets name of constant in string form

int ordinal() --rets index of the const as it appears in enum.--starts with 0

public String toString() : overridden to return name of the enum constant.

You can supply your own constructor/s to initialize the state(data member of enum types.

enum Direction {

// Enum types

EAST(0), WEST(180), NORTH(90), SOUTH(270);

// Constructor

private Direction(final int angle) {

this.angle = angle;

}

// Internal state

private int angle;

public int getAngle() {

return angle;

}

}

BUT u can't instantiate enums using these constructors , since they are implicitely private.

You can override toString BUT you can't override equals since it's declared as final method in enum.

NESTED CLASSES

1. The inner class(non-static nested) has access to all of the outer class's members, including those marked private , directly(without inst.)

BUT Outer class MUST make an instance of the inner class , to access it's members.

2. To instantiate an inner class, you must have a reference to an instance of the outer class.

syntax :

Instantiating a non-static nested class requires using both the outer inst and nested class names as follows:

BigOuter.Nested n = new BigOuter().new Nested();

3. Such Inner classes can't have static members.(Java SE 8 --allows static final data members)

About method-local inner classes

1.A method-local inner class is defined within a method of the enclosing class.

2.For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but after the class definition code.

3. A method-local inner class cannot use variables declared within the method

(including parameters) unless those variables are marked final or effectively final.

static nested classes

1.A static nested class is not an inner class, it's a top-level nested class.

2. You don't need an instance of the

outer class to instantiate a static nested class.

4.It cannot access non-static members of the outer class directly BUT can access static members of the outer class.

5. It can contain both static & non-static members.

6. JVM will not load any class's static init block -- until u actually refer to something from that class.

(Lazy loading) This is true for static nested classes too.

7. Instantiating a static nested class requires using Outer class name and instance of nested class names as follows:

Outer.Nested n = new Outer.new Nested();

WRAPPER CLASSES

Regarding wrapper classes

1. What's need of wrapper classes?

---1. to be able to add primitive types to growable collection(growable data structure eg -- LinkedList)

--- 2. wrapper classes contain useful api(eg --- parseInt,parseFloat....,isDigit,isWhiteSpace...)

2. What are wrappers? --- Class equivalent for primitive types

-- Inheritance hierarchy

java.lang.Object --- Character (char)

java.lang.Object --- Boolean

Object -- Number -- Byte,Short,Integer,Long,Float,Double

3. Constrs & methods --- for boxing & unboxing

boxing= conversion from prim type to the wrapper type(class type)

un-boxing = conversion from wrapper type to the prim type

eg

Integer(int data) --- boxing

Integer i1=new Integer(100);

//un-boxing

int data=i1.intValue();

Integer i1=100;//no err from JDK 1.5

sop(i1);

int data=1234;

i1++;//Integer--->int(auto unboxing), inc ,auto box

Object o=123.45;//auto-boxing(double--->Double)--up casted to Object

Number n1=true;//auto-box----X(up casted) to Number

Object o2=false;//auto box -- up casting

Double d1=1234;//auto boxing (int --->Integer) ---X--Double

4. JDK 1.5 onwards --- boxing &unboxing performed automatically by java compiler,when required. --- auto-boxing , auto-unboxing,

Day11

**GENERICS**

Generic syntax ---

Available from Java SE 5 onwards.

Represents Parameterized Types.

Can Create Generic classes, interfaces, methods and constructors.

In Pre-generics world , similar achieved via Object class reference.

Syntax -- similar to c++ templates (angle brackets)

eg : ArrayList<Emp> , HashMap<Integer,Account> .....

1. Syntax is different than C++ --for nested collections only.

2. NO code bloat issues unlike c++;

Advantages

Adds Type Safety to the code @ compile time

Meaning :

1. Can add type safe code where type-mismatch errors(i.e ClassCastExceptions) are detected at compile time.

2. No need of explicit type casting, as all casts are automatic and implicit.

A generic class means that the class declaration includes a type parameter.

eg --- class MyGeneric<T>

{

private T ref;

}

class MyGeneric<T,U> {...}

T ,U ---type --- ref type

eg : ArrayList<Emp>

Understand why generics with example.

eg : Create a Holder class , that can hold ANY data type (primitive/ref type)

DAY12

COLLECTIONS

List<E> features

1. List represents ordered collection --- order is significant(It remembers the order of insertion)

2. Allows null references

3. Allows duplicates

4. Supports index based operation

java.util.ArrayList<E> -- E -- type of ref.

1. ArrayList<E> -- constructor

API

ArrayList() -- default constructor. -- creates EMPTY array list object , with init capacity=10,size=0;

eg ---ArrayList<Integer> l1=new ArrayList<>();

1.5 1. ArrayList<E> -- constructor

API

public ArrayList(int capacity) -- -- creates EMPTY array list object , with init capacity=capacity,size=0;

eg ---ArrayList<Integer> l1=new ArrayList<>(100);

l1.add(1);.....l1.add(100);

l1.add(101);//capa=150 --as per JVM spec.

2. add methods

boolean add(E e) --- append

void add(int index,E e) --- insert

void addAll(Collection<E> e) -- bulk append operation

eg : l1 --- AL<Emp>

l1.addAll(.....);

AL,LL,Vector --- legal

HS,TS,LHS --legal

HM,LHM,TM --illegal --javac error

2.5 Retrieve elem from list

E get(int index)

index ranges from ---0 ---(size-1)

java.lang.IndexOutOfBoundsException

3. display list contents using --- toString

4. Attaching Iterator

Collection<E> interface method -- implemented by ArrayList

Iterator<E> iterator()

---places iterator BEFORE 1st element ref.

Iterator<E> i/f methods

boolean hasNext() -- rets true if there exists next element, false otherwise.

E next() --- returns the element next to iterator position

void remove() -- removes last returned element from iterator.

Limitation --- type forward only & can start from 1st elem only.

Regarding exceptions with Iterator/List

1. java.util.NoSuchElementException -- thrown whenever trying to access the elem beyond the size of list via Iterator/ListIterator

2. java.lang.IllegalStateException --- thrown whenever trying to remove elem before calling next().

3. java.util.ConcurrentModificationException-- thrown typically --- when trying to use same iterator/list iterator --after structrually modifying list(eg add/remove methods of list)

Above describes fail-fast behaviour of the Iterator/ListIterator

Exception while accessing element by index.

4. java.lang.IndexOutOfBoundsException -- thrown typically -- while trying to access elem beyond size(0---size-1) --via get

6. Attaching for-each = attaching implicit iterator.

Attaching ListIterator ---scrollable iterator or to beign iteration from a specific element -- List ONLY or list specific iterator.

ListIterator<E> listItearator() --places LI before 1st element

ListIterator<E> listItearator(int index) --places LI before specified index.

4. search for a particular element in list

boolean contains(Object o)

5. searching for 1st occurrence

use -- indexOf

int indexOf(Object o)

rets index of 1st occurrence of specified elem. Rets -1 if elem not found.

searching for last occurrence

use -- lastIndexOf

int lastIndexOf(Object o)

rets index of last occurrence of specified elem. Rets -1 if elem not found.

5.5

E set(int index,E e)

Replaces old elem at spepcified index by new elem.

Returns old elem

6. remove methods

E remove(int index) ---removes elem at specified index & returns removed elem.

boolean remove(Object o) --- removes element specified by argument , rets true -- if elem is removed or false if elem cant be removed.

Objectives in Integer list

0. Create ArrayList of integers & populate it.

1. check if element exists in the list.

2. disp index of 1st occurance of the elem

3. double values in the list --if elem val > 20

4. remove elem at the specified index

5. remove by elem. -- rets true /false.

NOTE :

For searching or removing based upon primary key , in List Implementation classes --- All search/remove methods (contains,indexOf,lastIndexOf,remove(Object o)) -- based upon equals method(of type of List eg --Account/Customer/Emp....)

For correct working

1. Identify prim key & create overloaded constr using PK.

eg : public Emp(int id) { this.id=id;}

2. Using PK , override equals for content equality.

Usage eg : ArrayList<Emp> emps=new AL<>();

emps.add(e1);//id=10

emps.add(e2);//id=20

emps.add(e3);//id=30

int index=emps.indexOf(20);//int ---> Integer --> Object (Integer)

Integer i=new Integer(20); // javac

//internally invokes equals : whose equals --Object | Integer | Emp | NOA

invokes equlas on Integer class

i.equals(e1) ---since it's incomptabile types --rets false

i.equals(e2) ---since it's incomptabile types --rets false

i.equals(e3) ---since it's incomptabile types --rets false

Thus : indexOf rets -1

sop(index);// -1

Solution :

Emp e=new Emp(20);

int index=emps.indexOf(e);

//internally invokes equals : whose equals --Object | Integer | Emp | NOA

invokes equals on Emp class

e.equals(e1) ---it's comptabile types BUT ids are different --rets false

e.equals(e2) --- it's comptabile types --ids are SAME --rets true

Thus : indexOf rets 1

sop(index);// 1

Objective --- Create simple List(ArrayList) of Account & test complete API

1.1

Create Empty Arraylist of Accounts

1.2 Accept a/c info from user till user types "stop" & populate AL.

1.2.1 -- Display AL content using for-each

1.3 Accept account id & display summary or error mesg

1.4 Accept src id , dest id & funds transfer.

1.5 Accept acct id & remove a/c --

1.6 Apply interest on all saving a/cs

1.7 Sort accounts as per asc a/c ids.

1.8 Sort accounts as per desc a/c ids.

1.9 Sort a/cs as per creation date -- w/o touching UDT

2.0 Sort a/cs as per bal

Sorting --- For sorting elements as per Natural(implicit i.e criteria defined within UDT class definition) ordering or Custom(explicit i.e criteria defined outside UDT , in a separate class or anonymus iner class)

Steps for Natural ordering

Natural Ordering is specified in generic i/f

java.lang.Comparable<T>

T -- UDT , class type of the object to be compared.

eg -- Emp,Account , Customer

I/f method

int compareTo(T o)

Steps

1. UDT must implement Comparable<T>

eg : public class Account implements Comparable<Account>

2. Must override method

public int compareTo(T o)

{

use sorting criteria to ret

< 0 if this < o,

=0 if this = o

> 0 if this > o

}

3.Use java.util.Collections class API

Method

public static void sort(List<T> l1)

l1 -- List of type T.

sort method internally invokes compareTo method(prog supplied) of UDT & using advanced sorting algorithm , sort the list elems.

Limitation of natural Ordering

Can supply only 1 criteria at given time & that too is embedded within UDT class definition

Instead keep sorting criteria external --using Custom ordering

Typically use -- Natural ordering in consistence with equals method.

Alternative is Custom Ordering(external ordering)

I/f used is --- java.util.Comparator<T>

T -- type of object to be compared.

Steps

1. Create a separate class (eg. AccountBalComparator) which implements Comparator<T>

eg

public class AccountBalComparator implements Comparator<Account>

2.Implement(override) i/f method -- to supply comparison criteria.

int compare(T o1,T o2)

Must return

< 0 if o1<o2

=0 if o1=o2

> 0 if o1 > o2

3. Invoke Collections class method for actual sorting.

public static void sort(List<T> l1,Comparator<T> c)

parameters

l1 --- List to be sorted(since List is i/f --- any of its implementation class inst. can be passed)

c - instance of the class which has implemented compare method.(or implemented Comparator)

Internally sort method invokes compare method from the supplied Comparator class instance.

More on generic syntax

Constructor of ArrayList(Collection<? extends E> c)

? -- wild card in generic syntax (denotes any unknown type)

--Added for supporting inheritance in generics.'

extends -- keyword in generics, to specify upper bound

? extends E -- E or sub type

Complete meaning --- Can create new populated ArrayList of type E , from ANY Collection(ArrayList,LinkedList,Vector,HashSet,LinkedhashSet,TreeSet) of type E or its sub type.

ArrayList<Emp> l1=new ArrayList<>();

l1.add(new Emp(1,"aa",1000);

l1.add(new Emp(2,"ab",2000);

ArrayList<Emp> l2=new ArrayList<>(l1);

sop(l2.size());

----------

HashSet<Emp> hs=new HashSet<>();

hs.add(new Emp(1,"aa",1000);

hs.add(new Emp(2,"ab",2000);

l2=new ArrayList<>(hs);

----

Vector<Mgr> v1=new Vector<>();

v1.add(new Mgr(....));

v1.add(new Mgr(....));

ArrayList<Emp> l2=new ArrayList<Mgr>(v1);

AL<Mgr> mgrs=new AL<>(hs);

Map API

HashMap<K,V> --

1. un-sorted(not sorted as per Natural ordering or custom ordering based criteria) & un-ordered(doesn't remember order of insertion) map implementation class.

2. No duplicate keys.

3. Guarantees constant time performance --- via 2 attributes --initial cpacity & load factor.

4. Allows null key reference(once).

5. Inherently thrd unsafe.

HashMap constrs

1. HashMap<K,V>() --- creates empty map , init capa = 16 & load factor .75

2. HashMap<K,V>(int capa) --- creates empty map , init capa specified & load factor .75

3.HashMap<K,V>(int capa,float loadFactor) --- creates empty map , init capa & load factor specified

4. HashMap constrcutor for creating populated map

HashMap(Map <? extends K,? extends V> m)

? -- wild card in generics, represents unknown type

extends -- represents upper bound

? extends K --- K or its sub type

? extends V -- V or its sub type.

Complete meaning -- Creates populated HM<K,V> from ANY map(ie. any Map imple class)

of type K or its sub type & V or its sub type.

eg : Suppose Emp <---- Mgr

HM<Integer,Emp> hm=new HM<>();

hm.put(1,e1);

hm.put(2,m1);

HM<Integer,Emp> hm2=new HM<>(hm);

sop(hm2);

LHM<Integer,Emp> lhm=new LHM<>(hm);//legal

HM<Integer,Mgr> hm3=new HM<Integer,Emp>(hm);//javac error

TM<Integer,Mgr> hm4=new TM<>();

hm4.put........

HM<Integer,Emp> hm5=new HM<>(hm4);

HM(Map<? extends K,? extends V>map)

put,get,size,isEmpty,containsKey,containValue,remove

Objective : Create AccountMap

Identify key & value type

create empty unsorted map(HashMap<K,V>) & populate the same

Disp all entries of HM ---can use only toString

1.get acct summary --- i/p --id o/p --- err / dtls

2.Withdraw --- specify Account id & Amt ---- o/p : update acct dtls if acct exists o.w err msg or exc

3.funds transfer ---

i/p sid,dest id, amt

4.remove --- account

i/p id

5.Apply interest on on saving type of a/cs.

or

display all accts created after date.

Attach for-each to map & observe.

Sort the map as per : asc order of accts Ids.

Sort the map as per : desc order of accts Ids

Sort the accts as per : balance

If map sorting involves key based sorting criteria --- can be sorted by converting into TreeMap

Constructors of TreeMap

1. TreeMap() -- Creates empty map , based upon natural ordering of keys

2. TreeMap(Map<? extends K,? extends V> map)

Creates populated map , based upon natural ordering of keys

3. TreeMap(Comparator<? super K> c)

Regarding generic syntax & its usage in TreeMap constructor.

<? super K>

? --- wild card --- any unknown type

super --- gives lower bound

K --- key type

? super K --- Any type which is either K or its super type.

TreeMap(Comparator<? super K> c) --- creates new empty TreeMap, which will sort its element as per custom ordering(i.e will invoke compare(...) of Key type )

<? extends K>

? -- any type or wild card

extends -- specifies upper bound

K -- key type

? extends K --- Any type as Key type or its sub type.

same meaning for <? extends V>

TreeMap(Map<? extends K,? extends V> m)

disp acct ids of all accounts ---impossible directly....(will be done by Collection view of map @ the end)

Apply interest to all saving type a/cs

difficult directly ---so get a collection view of the map & sort the same.

Limitations on Maps

1. Maps can be sorted as per key's criteria alone.

2. can't attach iterators/for-each(till JDK 1.7)/for

3 Maps can be searched as per key's criteria alone.

To fix --- get a collection view of a map (i.e convert map to collection)

API of Map i/f

1. To get set of keys asso. with a Map

Set<K> keySet();

2. To get collection of values from a map

Collection<V> values();

3. To get set of Entries(key & val pair) ---

entrySet

Set<Map.Entry> entrySet()

Methods of Map.Entry

K getKey()

V getValue()

7. conversion from collection to array

Object[] toArray() -- non generic version --- rets array of objects

T[] toArray(T[] type)

T = type of collection .

Rets array of actual type.

8. sorting lists --- Natural ordering creiteria

Using java.util.Collections --- collection utility class.

static void sort(List<E> l1) ---sorts specified list as per natural sorting criteria.

DAY13

**LINKED-LIST**

java.util.LinkedList<E>

Doubly-linked list implementation of the List and Deque interfaces.

It is an ordered collection and supports duplicate elements.

It stores elements in Insertion order.

It supports adding null elements.

It supports index based operations.

Typical use case -- stack or queue.

It does not implement RandomAccess interface.(ArrayList class does!)

So it represents sequential access list.

When we try to access an element from a LinkedList, searching that element starts from the beginning or end of the LinkedList based on whichever is closer to the specified index.(eg : list.get(i))

Structure of LinkedList --refer to diag.

It supports all of List API methods , as seen already in ArrayList.

Java LinkedList methods , inherited from Deque

The following methods are specific to LinkedList class which are inherited from Deque interface:

void addFirst(E e): Inserts the specified element at the beginning of this list.

void addLast(E e): Inserts the specified element at the end of this list.

E getFirst(): Retrieves, but does not remove, the first element of this list. This method differs from peekFirst only in that it throws an exception if this list is empty.

E getLast(): Retrieves, but does not remove, the last element of this list. This method differs from peekLast only in that it throws an exception if this list is empty.

E removeFirst(): Removes and returns the first element from this list.

E removeLast(): Removes and returns the last element from this list.

boolean offerFirst(E e): Inserts the specified element at the front of this list.

boolean offerLast(E e): Inserts the specified element at the end of this list.

E pollFirst(): Retrieves and removes the first element of this list, or returns null if this list is empty.

E pollLast(): Retrieves and removes the last element of this list, or returns null if this list is empty.

E peekFirst(): Retrieves, but does not remove, the first element of this list, or returns null if this list is empty.

E peekLast(): Retrieves, but does not remove, the last element of this list, or returns null if this list is empty.

Java LinkedList Usecases

Best Usecase scenario:-

When our frequently used operation is adding or removing elements in the middle of the List, LinkedList is the best class to use.

Why? Because we don’t need to do more shifts to add or remove elements at the middle of the list.

Worst Usecase scenario:-

When our frequently used operation is retrieving elements from list, then LinkedList is the worst choice.

Why? Because LinkedList supports only sequential access, does NOT support random access.

NOTE:-

LinkedList implements List, Deque, But it does NOT implement RandomAccess interface.

How to use LinkedList as a queue or stack ?

When a deque is used as a queue its FIFO (First-In-First-Out)Elements are added at the end of the deque and removed from the beginning.

Which method will you use ?????????

When a deque is used as stack its LIFO (Last-In-First-Out) (Preferred to the legacy Stack class.)

When a deque is used as a stack, elements are pushed and popped from the beginning of the deque.

Which method will you use ?????????

ARRAYLIST VS LINKED LIST

Difference between ArrayList and LinkedList in Java

ArrayList and LinkedList both implements List interface and their methods and results are almost identical. However there are few differences between them which make one better over another depending on the requirement.

ArrayList Vs LinkedList

1) Search: ArrayList search operation is pretty fast compared to the LinkedList search operation. get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n).

Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements doubly linked list which requires the traversal through all the elements for searching an element.

2) Deletion: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

Conclusion: LinkedList element deletion is faster compared to ArrayList.

Reason: LinkedList’s each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

3) Inserts Performance: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove.

4) Memory Overhead: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

There are few similarities between these classes which are as follows:

Both ArrayList and LinkedList are implementation of List interface.

They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order in which the elements got inserted into the List.

Both these classes are non-synchronized and can be made synchronized explicitly by using Collections.synchronizedList method.

The iterator and listIterator returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created, in any way except through the iterator’s own remove or add methods, the iterator will throw a ConcurrentModificationException).

When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance (O(1)) in LinkedList compared to ArrayList(O(n)). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get(index) method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

HASHSETS

Regarding Hashing based Data structures....(eg : HashSet,HashTable,HashMap)

Steps for Creating HashSet

1. Type class in HashSet must override : hashCode & equals method both in consistent manner.

Object class API

public int hashCode() --- rets int : which represents internal addr where obj is sitting on the heap(typically -- specific to JVM internals)

public boolean equals(Object ref) -- Object class rets true : iff 2 refs are referring to the same copy.

2. Rule to observe while overriding these methods

If 2 refs are equal via equals method then their hashCode values must be same.

eg : If ref1.equals(ref2) ---> true then ref1.hashCode() = ref2.hashCode()

Converse may not be mandatory.(i.e if ref1.equals(ref2) = false then its not mandatory that ref1.hashCode() != ref2.hashCode() : but recommended for better working of hashing based D.S)

String class , Wrapper classes , Date related classes have already folowed this contract.

Questions :

1. How does hashing based data structure ensure constant time performance?

If no of elements(size) > capacity \* load factor --- re-hashing takes place ---

New data structure is created --(hashtable) -- with approx double the original capacity --- HS takes all earlier entries from orig set & places them in newly created D.S -- using hashCode & equals. -- ensures lesser hash collisions.

2. Why there is a guarantee that a duplicate ref can't exist in yet another bucket ?

Answer is thanks to the contract between overriding of hashCode & equals methods

If two elements are the same (via equals() returns true when you compare them), their hashCode() method must return the same number. If element type violate this, then elems that are equal might be stored in different buckets, and the hashset would not be able to find elements (because it's going to look in the same bucket).

If two elements are different(i.e equals method rets false) , then it doesn't matter if their hash codes are the same or not. They will be stored in the same bucket if their hash codes are the same, and in this case, the hashset will use equals() to tell them apart.

MAPS

Map Overview (refer to the diagram "regarding Maps")

Map Implementation class

1. HashMap<K,V>

1.1 Constructors

1. HashMap()

2. HashMap(int initCapa)

3. HashMap(int initCapa,float loadFactor)

4. public HashMap(Map<? extends K,? extends V> m)

Meaning : Creates populated HashMap of type K,V from any Map (AL/LL/Vector : javac err

HS/LHS/TS : javac err

HM/LHM/TM : no error ) having generic type of K or its sub type & V or its sub type

Steps n API

0. Create new empy map to store account details

HashMap<Integer,BankAccount> hm=new HM<>();//size=0,capa=16,L.F=0.75

0.5 Create new account

Map i/f API

1. public V put(K key,V value)

Meaning : It will insert the new entry into map.If key already exists : it will replace old value by new value.

Rets : null in case of new entry or old value ref. in case of existing entry.

eg : sop(map.put(k1,v1));//null

sop(map.put(k2,v2));//null

sop(map.put(k3,v3));//null

sop(map.put(k1,v4));//v1

sop(map);

//which entries : { k1:v4,k2:v2 ,k3:v3}

2. public V putIfAbsent(K key,V value)

eg : sop(map.putIfAbsent(k1,v1));//null

sop(map.putIfAbsent(k2,v2));//null

sop(map.putIfAbsent(k3,v3));//null

sop(map.putIfAbsent(k1,v4));//v1

//which values(entries) : k1:v1 k2:v2 k3:v3

3. public void putAll(Map<? extends K,? extends V> m)

eg : map1.putAll(map2);

Meaning : It will copy all entries from map2 ----> map1

(put : replace)

4. public V get(Object key)

Rets value type of ref if key is found else rets null.

eg : map.get(k2) : v2

map.get(k10) : null

5. boolean containsKey(Object key)

Returns true if this map contains a mapping for the specified key , otherwise false;

eg : map.containsKey(k1) ---true

6. boolean containsValue(Object value)

Returns true if this map maps one or more keys to the specified value.

eg : map : k1:v1 k2:v2 k3:v3

map.containsValue(v3) ---- true

containsKey : O(1)

containsValue : O(n)

7. public V remove(Object K)

Tries to remove the entry(=mapping=key n value pair) if key is found --rets existing value ref.

Rets null if key is not found.;

eg : map : {k1:v1 k2:v2 k3:v3}

sop(map.remove(k2));//v2

sop(map);//k1:v1 k3:v3

sop(map.remove(k20));//null

sop(map);//k1:v1 k3:v3

How to overcome limitations of Map (can't iterate over map , can't search/sort/remove by any value based criteria)

Solution : Convert the map into its Collection view

1. How to extract key type refs from a Map ?

public Set<K> keySet()

eg : HM<Integer,BankAccount> hm=new HM<>();

add some a/cs

Set<Integer> keys =hm.keySet();//O(n)

2. How to get value type of references from a Map ?

public Collection<V> values();

eg : HM<Integer,BankAccount> hm=new HM<>();

added some a/cs ....

Collection<BankAccount> accts =hm.values();//O(n)

3. How to get key-value pair(entry) of references from a Map ?

Map : i/f

Nested i/f : Map.Entry<K,V> : Entry in a Map

public Set<Map.Entry<K,V>> entrySet();

4. Method of Map.Entry i/f

public K getKey()

public V getValue();

HASHMAP

How HashMap internally works in Java

Hash Map is one of the most used collection. It doesn't extend from Collection i/f.

BUT collection view of a map can be obtained using keySet,values or entrySet()

Internal Implementation

HashMap works on the principal of hashing.

What is hashing ?

Hashing means using some function or algorithm to map object data to some representative integer value.

Map.Entry interface ---static nested interface of Map i/f

This interface represents a map entry (key-value pair).

HashMap in Java stores both key and value object ref , in bucket, as an object of Entry class which implements this nested interface Map.Entry.

hashCode() -HashMap provides put(key, value) for storing and get(key) method for retrieving Values from HashMap.

When put() method is used to store (Key, Value) pair, HashMap implementation calls hashcode on Key object to calculate a hash that is used to find a bucket where Entry object will be stored.

When get() method is used to retrieve value, again key object is used to calculate a hash which is used then to find a bucket where that particular key is stored.

equals() - equals() method is used to compare objects for equality. In case of HashMap key object is used for comparison, also using equals() method Map knows how to handle hashing collision (hashing collision means more than one key having the same hash value, thus assigned to the same bucket. In that case objects are stored in a linked list (growable --singly linked)

Bucket term used here is actually an index of array, that array is called table in HashMap implementation. Thus table[0] is referred as bucket0, table[1] as bucket1 and so on.

HashMap uses equals() method to see if the key is equal to any of the already inserted keys (Recall that there may be more than one entry in the same bucket). Note that, with in a bucket key-value pair entries (Entry objects) are stored in a linked-list . In case hash is same, but equals() returns false (which essentially means more than one key having the same hash or hash collision) Entry objects are stored, with in the same bucket, in a linked-list.

In short , there are three scenarios in case of put() -

Using hashCode() method, hash value will be calculated. Using that hash it will be ascertained, in which bucket particular entry will be stored.

equals() method is used to find if such a key already exists in that bucket, if no then a new node is created with the map entry and stored within the same bucket. A linked-list is used to store those nodes.

If equals() method returns true, which means that the key already exists in the bucket. In that case, the new value will overwrite the old value for the matched key.

How get() methods works internally

As we already know how Entry objects are stored in a bucket and what happens in the case of Hash Collision it is easy to understand what happens when key object is passed in the get method of the HashMap to retrieve a value.

Using the key again hash value will be calculated to determine the bucket where that Entry object is stored, in case there are more than one Entry object with in the same bucket stored as a linked-list equals() method will be used to find out the correct key. As soon as the matching key is found get() method will return the value object stored in the Entry object.

In case of null Key

Since HashMap also allows null, though there can only be one null key in HashMap. While storing the Entry object HashMap implementation checks if the key is null, in case key is null, it always map to bucket 0 as hash is not calculated for null keys.

-----------------------------------------

HashMap changes in Java 8

Though HashMap implementation provides constant time performance O(1) for get() and put() method but that is in the ideal case when the Hash function distributes the objects evenly among the buckets.

But the performance may worsen in the case hashCode() used is not proper and there are lots of hash collisions. In case of hash collision entry objects are stored as a node in a linked-list and equals() method is used to compare keys. That comparison to find the correct key with in a linked-list is a linear operation so in a worst case scenario the complexity becomes O(n).

To address this issue in Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list but after the number of items in a bucket becomes larger than a certain threshold, the bucket will change from using a linked list to a balanced tree, this will improve the worst case performance from O(n) to O(log n).

More details

Hash collision degrades the performance of HashMap significantly. With this new approach, existing applications can expect performance improvements in case they are using HashMaps having large number of elements by simply upgrading to Java 8.

Hash collisions have negative impact on the lookup time of HashMap. When multiple keys end up in the same bucket, then values along with their keys are placed in a linked list. In case of retrieval, linked list has to be traversed to get the entry. In worst case scenario, when all keys are mapped to the same bucket, the lookup time of HashMap increases from O(1) to O(n).

Java 8 has come with the following improvements/changes of HashMap objects in case of high collisions.

The alternative String hash function added in Java 7 has been removed.

Buckets containing a large number of colliding keys will store their entries in a balanced tree instead of a linked list after certain threshold is reached.

Above changes ensure performance of O(log(n)) in worst case scenarios (hash function is not distributing keys properly) and O(1) with proper hashCode().

How linked list is replaced with binary tree?

In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

This JDK 8 change applies only to HashMap, LinkedHashMap and ConcurrentHashMap.

LAMBDA EXPRESSIONS

Lambda expressions

It's derived from lambda caculus.

It was a big change in calculus world , which gave tremedous ease in maths

Now the same concept is being used in programming languages.

1st language to use lambda

LISP

c , c++ , c# , scala , javascript , python

Finally in java also(Java SE 8 onwards)

Background

Java is an object-oriented language. With the exception of primitive data types, everything in Java is an object. Even an array is an Object. Every class creates instances that are objects. There is no way of defining just a function / method which stays in Java all by itself. There is no way of passing a method as argument or returning a method body for that instance.

i.e passing the behaviour was not possible till java 8.

It was slightly possible using anonymous inner classes --but that still required us to write a class !

What is lambda expression ?

Concise anonymous function which can be passed around

It has

1. list of params

2. body

3. return type.(optional)

Lambda expressions in Java is usually written using syntax (argument) -> (body). For example:

(type1 arg1, type2 arg2...) -> { body }

Following are some examples of Lambda expressions.

1.(int a, int b) -> { return a + b; }

OR can be reduced to

(int a, int b) -> a + b

OR further can be reduced to

(a,b) -> a+b

2. () -> System.out.println("Hello World")

3. s -> System.out.println(s)

4. () -> 42

5. () -> 3.1415

Above is just a syntax of lambda . But how to use them ?

Answer is ---You can use lambda expressions as targets of functional i/f reference.

----------------------

Why lambdas --

Easy way of passing a behaviour.

Till Java SE 7 , there was no way of passing a method as argument or returning a method body for that instance.

To enable this style of functional programming , lambdas are introduced.

How to pass a behaviour ?

------------------

What is a functional programming paradigm ?

A language where below features are supported.

Functions are treated as a first class citizens.

Meaning : You can

1.1 define anonymous functions

1.2 assign a function to a variable (function literal)

1.3 pass function as a parameter

1.4 return function as a return value

Why FP ?

1. To write more readable , maintainable , clean & concise code.

2. To use APIs easily n effectively.

3. To enable parallel processing

OOP uses imperative style of programming (where you will have to specify what's to be done & how --both) .

FP uses declarative style of programming (where you will just have to specify what's to be done

eg : Objective

Find out the average salary of emp from the specified dept.

How will you do it in imperative manner?

eg : List<Emp> l1=new AL<>();

l1.add(..);........

String dept=sc.next();

double total=0;

int num=0;

for(Emp e : l1)

if(e.getDept().equals(dept)) {

total += e.getSal();

num++;

}

sop(total/num);

OR

Vs

How to do it in declarative style ?

eg : List<Emp> l1=new AL<>();

l1.add(..);........

l1.stream().filter(e->e.getDept().equals(dept)).mapToDouble(Emp::getSal).average().getAsDouble()

---------------------------

Objective --

1. Perform ANY operation on 2 double values & return the result

eg --add/multiply/subtract/divide....

2. Convert from ANY src type to ANY dest type

eg : String ---> length

String ----> upper case string

celcius ---> fahrenheit (f=c\*1.8+32)

Student ---> GPA

number ---> square root

How do we declare just behaviour in java ?

Refer to Summable interface , implementation & its conversion to lambda.

Using interfaces

1. Create generic interface Converter<F,T> to specify single abstract method --convert , from F ---> T

2. Create a Tester class (with main method)

Add a static method to test the converter.

I/P -- 1. conversion source type(From)

2. conversion behaviour

O/P -- conversion result.(To)

public static <F,T> testConverter(F from, Converter<F,T> c)

{

return c.convert(from);

}

3. main(..) invokes this static method for testing Converter.

But what will be 2nd argument ?

Applying lambdas to refer to func i/f.

Objectives

1. Collections.sort

2. List -- forEach , removeIf

Main Differences between Lambda Expression and Anonymous class

1. One key difference between using Anonymous class and Lambda expression is the use of "this" keyword.

For anonymous class ‘this’ keyword resolves to anonymous class, whereas for lambda expression ‘this’ keyword resolves to enclosing class where lambda is written.

2. Another difference between lambda expression and anonymous class is in the way these two are compiled.

Java compiler compiles lambda expressions and convert them into private method of the class.

FUNCTIONAL PROGRAMMING

Regarding functional programming

What is functional programming ?

Functional programming is the way of writing s/w applications that uses only pure fuctions & immutable values.

Main concepts of FP are

1. Pure functions & side effects

2. Referential transperency

3. First class functions & higher order funcs.

4. Anonymous functions

5. Immutability

6. Recursion & tail recursion

7. Statements

8. Strict & Lazy evaluations

9. Pattern Matching

10 Closures

Why Functional Programming paradigm

Elegance and simplicity

Easier decomposition of problems

Code more closely tied to the problem domain

Through these , one can achieve :

Straightforward unit testing

Easier debugging

Simple concurrency

Functional programming In java

Functional Programming ( FP) is one of the type of programming pattern that helps the process of building application by using of higher order functions, avoiding shared state, mutable data

Functional programming vs OOP

Declarative vs Imperative :

Functional programming is a declarative pattern, meaning that the program logic is expressed without explicitly describing the flow control.Imperative programs spend lines of code describing the specific steps used to achieve the desired results — the flow control

Declarative programs remove the flow control process, and instead spend lines of code describing the data flow.

In Functional programming

Functions are treated as a first class citizens.

Meaning : You can

1.1 define anonymous functions

1.2 assign a function to a variable (function literal)

1.3 pass function as a parameter

1.4 return function as a return value

Why FP ?

1. To write more readable , maintainable , clean & concise code.

2. To use APIs easily n effectively.

3. To enable parallel processing

OOP uses imperative style of programming (where you will have to specify what's to be done & how --both) .

FP uses declarative style of programming (where you will just have to specify what's to be done

2. Functional interfaces

An interface which has exactly single abstract method(SAM) is called functional interface.

eg Runnable,Comparable,Comparator,Iterable,Consumer,Predicate,Supplier,Function...

Java SE 8 has introduced a new package for functional i/f

java.util.function

New annotation introduced -- @FunctionalInterface

(since Java SE 8)

Functional i/f references can be substituted by lambda expressions, method references, or constructor references.

Solve -- Is following valid functional interface ?

public interface A { double calc(int a,int b);} :

public interface B extends A {} :

public interface C extends A { void show();} :

public interface D {} -- Marker / empty / tag i/f :

public interface E extends A {default void show(){}} --

13. Addition of "default" keyword to add default method implementation , in interfaces.

Java 8 enables us to add non-abstract method implementations to interfaces by using the default keyword. This feature is also known as Extension Methods.

Why default keyword ?

1. To maintain backward compatibility with earlier Java SE versions

2. To avoid implementing new functionality in all implementation classes.

eg : Java added in Iterable<T> interface

default void forEach(Consumer<? super T> action) -- as a default method implementation

eg :

interface Formula {

double calculate(double a);//javac adds implicit keywords : public n abstract

//javac adds implicit keyword public

default double sqrt(double a,double b) {

return Math.sqrt(a+b);

}

}

Q : If you write an implementation class MyFormula

public class MyFormula imple Formula

{

.....

}

Which methods have to be implemented to avoid javac err?

1. calculate

2. sqrt

3. both

4. neither

Q : Can MyFormula class override the def. of sqrt ?

1 Display all elements of ArrayList

forEach

2. Create AL of integers

remove all odd numbers.

3. Create AL of emps

Remove underperforming employees (performance index < 7)

Dsiplay the list

4. Enter Java 8 Streams

1.Create int[] ---> IntStream & display its contents.

2. Create AL<Integer> , populate it.

Convert it to seq stream & display elems

Convert it to parallel stream & display elems

3. Create stream of ints between 1-100 & display all even elements.

(Hint : IntStream methods --range,filter,forEach)

4. Display all emp names from a particular dept , joined after specific date

(stream,filter,forEach)

5. Display sum of all even nos between 1-100 .

(stream , filter ,sum)

6. Display sum of salaries of all emps from a specific dept

7. Create a supplier of random numbers

eg :

Supplier<Double> randomValue = () -> Math.random();

// Print the random value using get()

System.out.println(randomValue.get());

8. Create a supplier of LocalDate & Time

eg : Supplier<LocalDateTime> s = () -> LocalDateTime.now();

LocalDateTime time = s.get();

System.out.println(time);

Supplier<String> s1 = () -> dtf.format(LocalDateTime.now());

String time2 = s1.get();

System.out.println(time2);

IO AND DEVICES

I/O handling

Desc of FileInputStream --- java.io.FileInputStream

bin i/p stream connected to file device(bin/char) -- to read data.

Desc of FileOutputStream --- java.io.FileOutputStream

bin o/p stream connected to file device(bin/char) -- to write data.

Desc of FileReader--- java.io.FileReader

char i/p stream connected to file device(char) -- to read data.

Desc of FileWriter--- java.io.FileWriter

char o/p stream connected to file device(char) -- to write data.

Objective --- Read data from text file in buffered manner.

1. java.io.FileReader(String fileName) throws FileNotFoundException

--- Stream class to represent unbuffered char data reading from a text file.

Has methods -- to read data using char/char[]

eg -- public int read() throws IOException

public int read(char[] data) throws IOException

Usage eg-- char[] data=new char[100];

int no= fin.read(data);

public int read(char[] data,int offset,int noOfChars) throws IOException

Usage eg-- char[] data=new char[100];

int no= fin.read(data,10,15);

eg -- 12 chars available

no=12;data[10]----data[21]

1.5 FileReader(File f) throws FileNotFoundException

java.io.File -- class represents path to file or a folder.

2. Improved version -- Buffered data read .

For char oriented streams--- java.io.BufferedReader(Reader r)

API of BR ---

String readLine() --- reads data from a buffer in line by line manner-- & rets null at end of Stream condition.

Objective -- Replace JDK 1.6 try-catch-finally BY JDK 1.7 try-with-resources syntax.

Meaning --- From Java SE 7 onwards --- Introduced java.lang.AutoCloseable -- i/f

It represents --- resources that must be closed -- when no longer required.

i/f method

public void close() throws Exception-- closing resources.

java.io --- classes -- have implemented this i/f -- to auto close resource when no longer required.

syntax of try-with-resources

try (//open one or multiple AutoCloseable resources)

{ ......

} catch(Exception e)

{

}

Objective ---To confirm device independence of Java I/O --- replace File device by Console

i.e --- Read data from console i/p --- in buffered manner till 'stop' & echo back it on the console.

required stream classes --- BR(ISR(System.in))

Alternative is --- use Scanner class.

Adv. of Scanner over above chain ----- contains ready-made parsing methods(eg --- nextInt,nextDouble.....)

But Scanner is not Buffered Stream

Can combine both approaches.(new Scanner(br.readLine())

Objective --- Combine scanner & buffered reader api --- to avail buffering + parsing api. ---

BufferedReader provides buffering BUT no simple parsing API. -- supplies br.readLine only

Scanner -- Can be attached to file directly

Constr -- Scanner(File f)

BUT no buffering .

How to use both?

Create BR br=new BR(new FR(...));

while ((s=br.readLine())!=null)

{

//scanner can be attached to string ---Scanner(String s)

Scanner sc=new Scanner(s);

// parse data using Scanner API --next,nextInt,nextBoolean

}

Overloaded constructor of FileReader(File f)

java.io.File ---- class represents path to file / folder

Regarding java.io.File -----

Does not follow stream class hierarchy, extends Object directly.

File class --- represents abstract path which can refer to file or folder.

Usage --- 1. To access/check file/folder attributes(exists,file or folder,read/w/exec permisssions,path,parent folder,create new empty file,create tmp files & delete them auto upon termination,mkdir,mkdirs,rename,move,size,last modified ,if folder---list entries from folder,filter entries)

Constructor ---

File (String path) ---

eg --- File f1=new File("abc.dat");

if (f1.exists() && f1.isFile() && f1.canRead())

...attach FileInputStream or FileReader

File (String path) ---

File class API --- boolean exists(),boolean isFile() , boolean canRead()

Objective --- Text File copy operation --- in buffered manner.

For writing data to text file using Buffered streams

java.io.PrintWriter --- char oriented buffered o/p stream --- which can wrap any device.(Binary o/p stream or Char o/p stream)

Constructors---

PrintWriter(Writer w) --- no auto flushing,no conversion, only buffering

PrintWriter(Writer w, boolean flushOnNewLine)--- automatically flush buffer contents on to the writer stream --upon new line

PrintWriter(OutputStream w) --- can wrap binary o/p stream -- buffering +conversion(char-->binary),no auto-flush option

PrintWriter(OutputStream w , boolean flushOnNewLine) ---

API Methods----print/println/printf same as in PrintStream class(same type as System.out)

Stream class which represents --- Char o/p stream connected to Text file. --- java.io.FileWriter

Constructor

FileWriter(String fileName) throws IOException -- new file will be created & data will be written in char format.

FileWriter(String fileName,boolean append) --- if append is true , data will be appended to existing text file.

-----------------

Collection & I/O

Objective ---

Items Inventory

Item -- code(String-Prim key),desc,category,quantity,price,shipmentDate

constr,toString.

Create suitable collection of Items(HashMap) --- sort map as per desc item code ,& store sorted item dtls in 1 text file .

NOTE : individual item rec MUST be written on separate line.

Sort items as shipment Date & store sorted dtls in another file . Before exiting ensure closing of data strms .

(buffered manner)

Objective -- Restore collection of items created in above requirement ---in form of HashMap . -- buffering is optional.

Objective --- using Binary file streams.

Classes --- FileInputStream -- unbuffered bin i/p stream connected to bin file device.

FileOutputStream --unbuffered bin o/p stream connected to bin file device.

But these classes --- dont provide buffering & have only read() write() methods in units of bytes/byte[]

API of InputStream class

1. int read() throws IOException

Will try to read 1 byte from the stream.

Data un-available method blocks.

Returns byte--->int to caller.

eg -- int data=System.in.read();

2. int read(byte[] bytes) throws IOException

Will try to read data from underlying stream.

Data un-available -- method blocks.

Rets actual no of bytes read.

eg :

byte[] bytes=new byte[100];

int no=System.in.read(bytes);

no data --method blocks.

10 bytes available -no =10;bytes[0]-----bytes[9]

110 bytes available -- no=100;bytes[0]....bytes[99]

3. int read(byte[] bytes,int offset,int maxNoOfBytes) throws IOException

Will try to read data from underlying stream.

Data un-available -- method blocks.

Rets actual no of bytes read.

eg :

byte[] bytes=new byte[100];

int no=System.in.read(bytes,10,15);

no data --BLOCKS

5 bytes available --no=5;bytes[10].....bytes[14]

110 bytes available -- no=15;bytes[10]..bytes[24]

4. int available() throws IOException

Returns no of available bytes in the stream

no data ---DOESN't block -- rets 0.

Important API of OutputStream

1. public void write(int byte) throws IOException

2. public void write(byte[] bytes) throws IOException

3. public void write(byte[] bytes,int offset,int maxNo) throws IOException

bytes[offset].....bytes[offset+maxNo-1] -- written out to stream

4. void flush() throws IOException

5. void close() throws IOException

Using BIS(BufferedInputStream) -- enables buffering BUT doesn't provide any advanced API(ie. read(), read(byte[]), read(byte[] b,int off,int len) . Same is true with BOS.(BufferedOutputStream)

Objective ---

Create Customer/Account based collection. Sort if reqd.

Store Sorted collection to bin file in buffered manner --

& re-store the same.

Use advanced streams in such cases ---

Mixed Data streams

java.io.DataOutputStream ---implements DataOutput i/f

(converter stream ) prim types / string ---> binary

Constructor -- DataOutputStream (OutputStream out)

API ---

public void writeInt(int i) throws IOExc

public void writeChar(char i) throws IOExc

public void writeFloat,writeDouble.....

For Strings

public void writeUTF(String s) throws IOExc ---uses Modified UTF 8 convention

or

public void writeChars(String s) throws IOExc --- uses UTF16 convention

eg : Items Inventory

Item -- code(String-Prim key),desc,category,quantity,price,shipmentDate

constr,toString.

Objective ---

Customer data is already stored in bin file.

Read customer data from Bin file --- in buffered manner & upload the same in HM .display customer details.

Stream class --- java.io.DataInputStream -- implements DataInput

Conversion stream(converts from bin ---> prim type or String)

Constructor

DataInputStream(InputStream in)

API Methods

public int readInt() throws IOException

public double readDouble() throws IOException

public char readChar() throws IOException

public String readUTF() throws IOException(must be used with writeUTF)

public String readChars() throws IOException(must be used with writeChars)

Most Advanced streams ---

Binary streams which can read/write data from/to binary stream in units of Object/Collection of Object refs (i.e Data Transfer Unit = Object/Collection of Objects)

Stream Class for writing Objects to bin. stream

java.io.ObjectOutputStream implements DataOutput,ObjectOutput

Description --- ObjectOutputStream class performs serialization.

serialization= extracting state of object & converting it in binary form.

state of object = non-static & non-transient data members

Constructor

ObjectOutputStream(OutputStream out)

out--- dest Binary o/p stream --- where serialized data stream has to be sent.

API methods ---

public void writeInt(int i) throws IOExc

public void writeChar(char i) throws IOExc

public void writeFloat,writeDouble.....

For Strings

public void writeUTF(String s) throws IOExc ---uses Modified UTF 8 convention

public void writeObject(Object o) throws IOException,NotSerializableException

De-serialization---- conversion or re-construction of Java objs from bin stream.

java.io.ObjectInputStream --- performs de-serialization.--- implements DataInput,ObjectInput

Constructor --- ObjectInputStream(InputStream in)

API methods ---

readInt,readShort,readUTF,readChars..... +

public Object readObject() throws IOException

Objective --- attach OIS to the bin file using FIS & display customer data.

Objective :

Confirming concepts of serialization & de-serialization

Emp -- int id, String name,double salary,Address adr;

Address -- String state,city,street.

Objective -- Understanding Set & its implementation classes

HashSet -- based upon hashing algorithm

More involved scenario.

(store customer info & Items to be purchased)

Data members - int no,Customer info, AL<Item>, Date creationDate

STREAMS

What is a Stream?

A sequence of elements from a source that supports data processing operations.

Sequence of elements - Like a collection, a stream provides an interface to a sequenced set of values of a specific type.

Source - Streams refer to collections, arrays, or I/O resources.

Data processing operations - Supports common operations from functional programming languages.

e.g. filter, map, reduce, find, match, sort etc

They have nothing to do with java.io -- InputStream or outputstream

The Streams also support Pipelining and Internal Iterations. The Java 8 Streams are designed in such a way that most of its stream operations returns Streams only. This help us creating chain of various stream operations. This is called as pipelining. The pipelined operations looks similar to a sql query.(or Hibernate Query API)

Concurrency is IMPORTANT. But it comes with a learning curve.

So , Java 8 goes one more step ahead and has developed a Streams API which allows us to use multi cores easily.

Parallel processing = divide a larger task into smaller sub tasks (forking), then processing the sub tasks in parallel and then combining the results together to get the final output (joining).

Java 8 Streams API provides a similar mechanism to work with Java Collections.

The Java 8 Streams concept is based on converting Collections to a Stream (or arrays to a stream) , processing the elements in parallel and then gathering the resulting elements into a Collection.

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both non-interfering and stateless. What does that mean?

A function is non-interfering when it does not modify the underlying data source of the stream, e.g.

List<String> myList =Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList.stream().filter(s -> s.startsWith("c")).map(String::toUpperCase) .sorted()

.forEach(System.out::println);

In the above example no lambda expression does modify myList by adding or removing elements from the collection.

A function is stateless when the execution of the operation is deterministic,

e.g. in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

API

The starting point is java.util.stream.Stream i/f

Different ways of creating streams

1. Can be created of any type of Collection (Collection, List, Set):

java.util.Collection<E> API

1.1 default Stream<E> stream()

1.2 public default Stream<E> parallelStream()

NOTE that Java 8 streams can’t be reused, will raise IllegalStateException

2. Stream of Array

How to create stream from an array?

Arrays class API

public static <T> Stream<T> stream(T[] array)

Returns a sequential Stream with the specified array as its source.

3. Can be attached to Map ,via entrySet method.

Refer to CreateStreams.java

4. To create streams out of three primitive types: int, long and double.

As Stream<T> is a generic interface , can't support primitives.

So IntStream, LongStream, DoubleStream are added.

API of java.util.stream.IntStream

4.1 static IntStream of(int... values)

Returns a sequential ordered stream whose elements are the specified values.

4.2 static IntStream range(int startInclusive,int endExclusive)

Returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.

4.3 static IntStream rangeClosed(int startInclusive,int endInclusive)

Returns a sequential ordered IntStream from startInclusive (inclusive) to endInclusive (inclusive) by an incremental step of 1.

5. To perform a sequence of operations over the elements of the data source and aggregate their results, three parts are needed – the source, intermediate operation(s) and a terminal operation.

6.java.util.stream.Stream<T> i/f API

6.1 Stream<T> skip(long n)

Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream(stateful intermediate operation)

6.2 map

<R> Stream<R> map(Function<? super T,? extends R> mapper)

Returns a stream consisting of the results of applying the given function to the elements of this stream(intermediate stateless operation)

mapToInt

IntStream mapToInt(ToIntFunction<? super T> mapper)

Returns an IntStream consisting of the results of applying the given function to the elements of this stream.

6.3 filter

Stream<T> filter(Predicate<? super T> predicate)

Returns a stream consisting of the elements of this stream that match the given predicate.(intermediate stateless operation)

ref : StreamAPI1.java

-------------------

7. Confirm lazyness of streams.

Intermediate operations are lazy. This means that they will be invoked only if it is necessary for the terminal operation execution.

ref : LazyStreams.java

8. Reduce operation

Readymade methods of IntStream

count(), max(), min(), sum(),average()

9. Customized reduce operation

ref : ReduceStream.java

10 collect

Reduction of a stream can also be executed by another terminal operation – the collect() method.

eg : StreamCollect.java

Good examples in java.util.stream.Collectors -api docs.

-----------------------

Details ---

1. Streams are functional programming design pattern for processing sequence of elements sequentially or in parallel.(a.k.a Monad in functional programming)

2. Stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements

3. Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations .

Terminal operations are either void or return a non-stream result.

4. They can't be reused.

5. Collections vs Streams:

Collections are in-memory data structures which hold elements within it. Each element in the collection is computed before it actually becomes a part of that collection. On the other hand Streams are fixed data structures which computes the elements on-demand basis.

The Java 8 Streams -- lazily constructed Collections, where the values are computed when user demands for it.

Actual Collections behave absolutely opposite to it and they are set of eagerly computed values (no matter if the user demands for a particular value or not).

METHOD REFERENCE

Method References in Java 8

Method reference is a shorthand , more readable notation of a lambda expression to call a method.

Can all lambda expressions be concised into method reference ? NO

eg :

If your lambda expression is like this:

s -> System.out.println(s)

then you can replace it with a method reference like this:

(since we are directly calling an existing method in a lambda expression , we can refer to the method itself)

System.out::println

The :: operator is used in method reference to separate the class or object from the method name

Four types of method references

1. Method reference to an instance method of an object – object::instanceMethod

2. Method reference to a static method of a class – Class::staticMethod

3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod

4. Method reference to a constructor – Class::new

1. Method reference to an instance method of an object

@FunctionalInterface

interface MyInterface{

void display();

}

public class Example {

public void myMethod(){

System.out.println("Instance Method");

}

public static void main(String[] args) {

Example obj = new Example();

// lambda expression :

//What is this lambda expression doing ? It's simply calling an existing method. In this case can it be further shortened by a method ref?

MyInterface ref = () -> obj.myMethod();

ref.display();

// Can you replace lambda expression by Method reference here ? (using the object of the class) : YES

MyInterface ref2 = obj::myMethod;

ref2.display();

}

}

2. Method reference to a static method of a class

import java.util.function.BiFunction;

class Multiplication{

public static int multiply(int a, int b){

return a\*b;

}

}

public class Example {

public static void main(String[] args) {

BiFunction<Integer, Integer, Integer> product = Multiplication::multiply;

int pr = product.apply(11, 5);

System.out.println("Product of given number is: "+pr);

}

}

3. Method reference to an instance method of an arbitrary object of a particular type

import java.util.Arrays;

public class Example {

public static void main(String[] args) {

String[] stringArray = { "aa", "bb", "cc", "dd", "ee"};

/\* Method reference to an instance method of an arbitrary

\* object of a particular type

\*/

//Arrays.sort(stringArray, (s1,s2)->s1.compareTo(s2));

Arrays.sort(stringArray, String::compareTo);

Arrays.stream(stringArray).forEach(System.out::println);

}

}

4. Method reference to a constructor

@FunctionalInterface

interface MyInterface{

Hello display(String say);

}

class Hello{

public Hello(String say){

System.out.print(say);

}

}

public class Example {

public static void main(String[] args) {

//Method reference to a constructor

MyInterface ref = Hello::new;

ref.display("Hello World!");

}

}

4.1

@FunctionalInterface

interface MyFunctionalInterface {

Student getStudent();

}

class Student {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Following example uses constructor reference.

public class ConstructorReferenceDemo {

public static void main(String[] args) {

MyFunctionalInterface ref = Student::new;

Supplier<Student> s1 = Student::new;// Supplier Example

Supplier<Student> s2 = () -> new Student();// equals to above line

System.out.println(ref.getStudent());//Student class toString() call

System.out.println(s1.get());//Student class toString() call

}

}

SERIALIZATION-DESERIALIZATION

What is the need of ObjectInputStream & ObjectOutputStream ?

To achieve Persistence.

Persistence=Saving the state of the java object in permanent manner.

In the absence of Object streams, if you want to persist(save in permanent manner) state of objects or application data in binary manner --- prog has to convert all data to binary & then only it can be written to streams.

Object streams supply ready made functionality for the same.

Stream Class for writing Objects to bin. stream

java.io.ObjectOutputStream implements DataOutput,ObjectOutput

Description --- ObjectOutputStream class performs serialization.

serialization= extracting state of object & converting it in binary form.

(Details --Serialization literally refers to arranging something in a sequence. It is a process in Java where the state of an object is transformed into a stream of bits. The transformation maintains a sequence in accordance to the metadata supplied)

state of object = non-static & non-transient data members

Constructor

ObjectOutputStream(OutputStream out)

out--- dest Binary o/p stream --- where serialized data stream has to be sent.

API methods ---

public void writeInt(int i) throws IOExc

public void writeChar(char i) throws IOExc

public void writeFloat,writeDouble.....

For Strings

public void writeUTF(String s) throws IOExc ---uses Modified UTF 8 convention

+

public void writeObject(Object o) throws IOException,NotSerializableException

De-serialization---- conversion or re-construction of Java objs from bin stream.

java.io.ObjectInputStream --- performs de-serialization.--- implements DataInput,ObjectInput

Constructor --- ObjectInputStream(InputStream in)

API methods ---

readInt,readShort,readUTF,readChars..... +

public Object readObject() throws IOException,ClassNotFoundException,InvalidClassException

Serialization/De-serialization

Ability to write or read a Java object to/from a binary stream

Supported since JDK 1.1

Saving an object to persistent storage(current example -- bin file later can be replaced by DB or sockets) is called persistence

Java provides a java.io.Serializable interface for checking serializability of java classes.(object)

Meaning --- At the time of serialization(writeObject) or de-serialization(readObject) --- JVM checks if the concerned object is Serializable(i.e has it implemented Serializable i/f) --if yes then only proceeds , otherwise throws Exception ---java.io.NotSerializableException

Serializable i/f has no methods / data members and is a marker(tag) interface. Its role is to provide a run time marker for serialization.

Details

What actually gets serialized?

When an object is serialized, mainly state of the object(=non-static & non-transient data members) are preserved.

----------------------------------------------------

If a data member is an object(ref) , data members of the object are also serialized if that objects class is serializable

eg : If Product class HAS - A reference of ShippingAddress

The tree of objects data, including these sub-objects constitutes an object graph

eg : HM<String,Product> hm .....

out.writeObject(hm);

HM -- String --Product (id,name,price,qty,category +shippingDetails)

If a serializable object contains reference to non-serializable element, the entire serialization fails

If the object graph contains a non-serializable object reference, the object can still be serialized if the non-serializable reference is marked transient

Details --- transient is a keyword in java.

Can be applied to data member.(primitive as well as ref types)

transient implies ---skip from serialization.(meant for JVM)

During de-serialization ---transient(or even static) members are initialized to def values.

Usage -- To persist --partial state of the serializable object

---------------------------------------

If super-class is serializable, then sub-class is automatically serializable.

If super-class is NOT serializable --- super class must have a default constructor (otherwise InvalidClassException is thrown by JVM during de serilaization)

sub-class developer has to explicitely write the state of super-class.

What happens during deserialization?(in.readObject())

When an object is deserialized, the JVM tries to bring the object back to life by making a new object on the heap that has the same state the serialized object had at the time it was serialized.

1. (Class.forName("com.app.core.Account")--class loading purpose,

1.5 Matches incoming Serial version UID with the computed one

If matches --continues to steps 2.

Otherwise --- InvalidClassException is thrown.

2. If JVM comes across any non serializable super class , having no def constr --- InvalidClassExc is thrown.

Otherwise continues

Class.newInstance() or similar reflection API -- EMPTY/BLANK object is created on heap.

3. setting state of the object from bin stream)

The static/transient variables, which come back have either null (for object references) or as default primitive values.

4. Constructor of serializable class does not get called during de-serialization.

why ?

Think -- what is the need of constructor?

Constructor initializes the object variables with either default values or values which is assigned inside constructor. BUT we want to initialize the state of the object from binary stream.

---------------

What are pre-requisites for de-serialization?

Byte codes (.class file) for entire object graph to be de-serialized + Bin data stream containing state.

--------------------

Details

Java Deserializing process says, "For serializable objects, the no-arg constructor for the first non-serializable supertype is run."

It means during deserialization process, JVM checks the inheritance class hierarchy of instance in process.

It checks, if the Class instance in process has implemented Serializable interface, If yes, then JVM will check Parent class(If any) of the instance to see if Parent has implemented Serializable interface, If yes, then JVM continues to check all its Parents upwards till it encounters a class which doesn't implement Serializable interface. If all inheritance hierarchy of instance has implemented Serializable interface as one shown above then JVM will end up at default extended Object class which doesn't implemented Serializable interface. So it will invoke a default constructor of Object class.

If in between searching the super classes, any class is found non-serializable then its default constructor will be used . If any super class of instance to be de-serialized is non-serializable and also does not have a default constructor then the java.io.InvalidClassException is thrown by JVM.

So till now we got the instance located in memory using one of superclasss default constructor. Note that after this no constructor will be called for any class. After executing super class constructor, JVM read the byte stream and use instances meta data to set type information and other meta information of instance.

After the blank instance is created, JVM first set its static fields and then invokes the default readObject() method (if its not overridden, otherwise overridden method will be called) internally which is responsible for setting the values from byte stream to blank instance. After the readObject() method is completed, the deserialization process is done and you are ready to work with new deserialized instance.

--------------------------

What is serialversion UID?

The serialVersionUID is a universal version identifier for a Serializable class. Deserialization uses this number to ensure that a loaded class corresponds exactly to a serialized object. If no match is found, then an InvalidClassException is thrown.

How to generate ?

serialver F.Q class name(for a class that imple. Serializable)

eg : serialver java.util.HashMap

Details

Each time an object is serialized, the object (including every object in its graph) is 'stamped' with a version ID number for the object's class. The ID is called the serialVersionUlD, and it's computed based on information about the class structure(class constructors,implemented interfaces,data members).

As an object is being deserialized, if the class has changed since the object was serialized, the class could have a different serialVersionUID, and deserialization will fail.(java.lang.InvalidClassException).

Since different java compilers or even different versions of java compilers can produce different serial version UID , its always recommended , that a programmer should add UID even in the 1st version of class & modify it whenever the class is modified substantially.

examples of incompatible changes

-- Deleting fields , Moving classes up or down the hierarchy ,changing a non-static field to static or a non-transient field to transient ,changing the declared type of a primitive field ....

examples of compatible changes

adding fields,adding classes,adding Serializable,modifying access specifer of the field....

-----------------------------------

Serialization format overview

Contents of serialized binary stream

It has all the information about the instance which was serialized by serialization process.

This information includes

class’s meta data

type information of instance fields

values of instance fields as well.

This same information is needed when object is re-constructed back to a new object instance. While deserializing an object, the JVM reads its class metadata from the stream of bytes which specifies whether the class of an object implements either ‘Serializable’ or ‘Externalizable’ interface.

Detailed format

1. Magic no.

2. Serialization format version no.

3. Class description -- class name,serial version uid,desc of data members

4. State of the object.(non static & non transient data members)

-----------

Limitations

1. Java technology only

2. Difficult to maintain in case of changing class format

3. May lead to security leaks.

-----------------------

Important facts of serialization n deserialization

1.

Transient and static fields are ignored in serialization. After de-serialization transient fields and non-final static fields will be inited to default values. Final static fields still have values since they are part of the class data.

2.

ObjectOutputStream.writeObject(obj) and ObjectInputStream.readObject() are used in serialization and de-serialization.

3.

During serialization, you need to handle IOException; during de-serialization, you need to handle IOException and ClassNotFoundException. So the de-serializaed class type must be in the classpath.

4.

Uninitialized non-serializable, non-transient instance fields are tolerated. When adding "private Address adr; no error during serialization.

But , private Address adr = new Address(); will cause exception:

Exception in thread "main" java.io.NotSerializableException: com.app.core.Address

5. Serialization and de-serialization can be used for copying and cloning objects. It is slower than regular clone, but can produce a deep copy very easily.

6. If you need to serialize a Serializable class Employee, but one of its super classes is not Serializable, can Employee class still be serialized and de-serialized?

The answer is yes, provided that the non-serializable super-class has a no-arg constructor, which is invoked at de-serialization to initialize that super-class.

What will be the state of data members?

Sub class (serializable) data members will have the restored state & super class(non serializable) data members will have def inited state

7. You must be careful while modifying a class implementing java.io.Serializable. If class does not contain a serialVersionUID field, its serialVersionUID will be automatically generated by the compiler(using serialver tool). Different compilers, or different versions of the same compiler, will generate potentially different values.

Computation of serialVersionUID is based on not only fields, but also on other aspect of the class like implements clause, constructors, etc. So the best practice is to explicitly declare a serialVersionUID field to maintain backward compatibility. If you need to modify the serializable class substantially and expect it to be incompatible with previous versions, then you need to increment serialVersionUID to avoid mixing different versions.

8. Important differences between Serializable and Externalizable

8.1

If you implement Serializable interface , automatically state of the object gets seriliazed. BUT if u implement Externalizable i/f -- you have to explicitly mention which fields you want to serialize.

8.2

Serializable is marker interface without any methods. Externalizable interface contains two methods: writeExternal() and readExternal().

8.3

Default Serialization process will take place for classes implementing Serializable interface. Programmer defined Serialization process for classes implementing Externalizable interface.

8.4

Serializable i/f uses java reflection to re construct object during de-serialization and does not require no-arg constructor. But Externalizable requires public no-arg constructor.

THREADS

Thread related API

Starting point

1. java.lang.Runnable --functional i/f

SAM (single abstract method) -- public void run()

Prog MUST override run() -- to supply thread exec. logic.

2. java.lang.Thread --class -- imple . Runnable

It has imple. run() -- blank manner.

3. Constrs of Thread class in "extends" scenario

3.1 Thread() -- Creates a new un-named thrd.

3.2 Thread(String name) -- Creates a new named thrd.

4. Constrs of Thread class in "implements" scneario

4.1 Thread(Runnable instance) -- Creates a new un-named thrd.

4.2 Thread(Runnable instance,String name) -- Creates a new named thrd.

Methods of Thread class

1. public void start() -- To cause transition from NEW --> RUNNABLE

throws IllegalThreadStateException -- if thrd is alrdy runnable or dead.

2. public static void yield() -- Requests the underlying native scheduler to release CPU & enters rdy pool.

Use case -- co operative multi tasking(to allow lesser prio thrds to access processor)

3. public void setName(String nm)

4. public String getName()

5. Priority scale -- 1---10

Thread class consts --MIN\_PRIO=1 , MAX\_PRIO=10 , NORM\_PRIO =5

public void setPriority(int prio)

6. public static Thread currentThread() -- rets invoker(current) thrd ref.

7. public String toString() -- Overrides Object class method , to ret

Thread name,priority,name of thrd grp.

8.public static void sleep(long ms) throws InterruptedException

Blocks invoker thread till specified msecs.

9. public void join() throws InterruptedException

Blocking method(API)

--Causes the invoker thread to block till specified thread gets over.

eg : t1 & t2

t1's run()

{

.....

t2.join();//who is waiting for whom for which purpose ? : t1 is waiting for t2 : to complete exec

....

}

t2's run()

{

//some B.L :read data from file

}

join method can be used effectively to avoid orphan threads

main has to wait for child thrds to complete exec

How ?

In main(..)

t1.join();

t2.join();

10 public void join(long ms) throws InterruptedException

eg : In main method

t1.join(10000);//main is waiting for t1 to finish exec: upto max 10 sec

//t1 gets over after 2 secs : main un blocks

//If t1 doesn't get over within 10 secs : main will be blocked for 10 sec n auto un block.

--Causes the invoker thread to block till specified thread gets over OR tmout elapsed

11. public void interrupt() -- interrupts(un blocks ) the threads blocked on ---sleep/join/wait

Methods of Object class (Use Case : Inter thread communication)

1. public final void wait() throws InterruptedException,IllegalMonitorStateException

Meaning -- Forces the invoker thread to release processor & monitor & wait outside .

Trigger for InterruptedException

Some other thread sends interrupt signal to the waiting thread.

Trigger for IllegalMonitorStateException

If the invoker thread is not an owner of the monitor

(i.e if its invoking neither a synched method nor a block)

2. public final void notify() throws IllegalMonitorStateException

Meaning -- Un blocks (wakes up) exactly 1 thread , which has invoked wait on the same object's monitor.

May raise IllegalMonitorStateException --if the current thread is not the owner of a lock.

3. public final void notifyAll() throws IllegalMonitorStateException

Un blocks ALL waiting threads , on the same object's monitor.

THREAD API

Thread related API

Starting point

1. java.lang.Runnable --functional i/f

SAM (single abstract method) -- public void run()

Prog MUST override run() -- to supply thread exec. logic.

2. java.lang.Thread --class -- imple . Runnable

It has imple. run() -- blank manner.

3. Constrs of Thread class in "extends" scenario

3.1 Thread() -- Creates a new un-named thrd.

3.2 Thread(String name) -- Creates a new named thrd.

4. Constrs of Thread class in "implements" scneario

4.1 Thread(Runnable instance) -- Creates a new un-named thrd.

4.2 Thread(Runnable instance,String name) -- Creates a new named thrd.

Methods of Thread class

1. public void start() -- To cause transition from NEW -- RUNNABLE

throws IllegalThreadStateException -- if thrd is alrdy runnable or dead.

2. public static void yield() -- Requests the underlying native scheduler to release CPU & enters rdy pool.

Use case -- co operative multi tasking(to allow lesser prio thrds to access processor)

3. public void setName(String nm)

4. public String getName()

5. Priority scale -- 1---10

Thread class consts --MIN\_PRIO=1 , MAX\_PRIO=10 , NORM\_PRIO =5

public void setPriority(int prio)

6. public static Thread currentThread() -- rets invoker(current) thrd ref.

7. public String toString() -- Overrides Object class method , to ret

Thread name,priority,name of thrd grp.

8.public static void sleep(long ms) throws InterruptedException

Blocks invoker thread till specified msecs.

9. public void join() throws InterruptedException

Blocking method(API)

--Causes the invoker thread to block till specified thread gets over.

eg : t1 & t2

t1's run()

{

.....

t2.join();//who is waiting for whom for which purpose ? : t1 is waiting for t2 : to complete exec

....

}

t2's run()

{

//some B.L :read data from file

}

join method can be used effectively to avoid orphan threads

main has to wait for child thrds to complete exec

How ?

In main(..)

t1.join();

t2.join();

10 public void join(long ms) throws InterruptedException

eg : In main method

t1.join(10000);//main is waiting for t1 to finish exec: upto max 10 sec

//t1 gets over after 2 secs : main un blocks

//If t1 doesn't get over within 10 secs : main will be blocked for 10 sec n auto un block.

--Causes the invoker thread to block till specified thread gets over OR tmout elapsed

11. public void interrupt() -- interrupts(un blocks ) the threads blocked on ---sleep/join/wait

Methods of Object class (Use Case : Inter thread communication)

1. public final void wait() throws InterruptedException,IllegalMonitorStateException

Meaning -- Forces the invoker thread to release processor & monitor & wait outside .

Trigger for InterruptedException

Some other thread sends interrupt signal to the waiting thread.

Trigger for IllegalMonitorStateException

If the invoker thread is not an owner of the monitor

(i.e if its invoking neither a synched method nor a block)

2. public final void notify() throws IllegalMonitorStateException

Meaning -- Un blocks (wakes up) exactly 1 thread , which has invoked wait on the same object's monitor.

May raise IllegalMonitorStateException --if the current thread is not the owner of a lock.

3. public final void notifyAll() throws IllegalMonitorStateException

Un blocks ALL waiting threads , on the same object's monitor.

THREAD START

What happens when you call start on NEW Thread ?

It internally invokes a native method(not written in java) start0()

Its invocation will --

1. cause a new native thread-of-execution to be created (by native OS)

2. cause the run method to be invoked on that thread.

THREAD RACE CONDITION

A race condition is a special condition that may occur inside a critical section. A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition.

Race condition means that the threads are racing through the critical section, and that the result of that race impacts the result of executing the critical section.

Critical Sections

Running more than one thread inside the same application does not by itself cause problems. The problems arise when multiple threads access the same resources. For instance the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files.

In fact, problems only arise if one or more of the threads write to these resources. It is safe to let multiple threads read the same resources, as long as the resources do not change.

Here is a critical section Java code example that may fail if executed by multiple threads simultaneously:

public class Counter {

protected long count = 0;

public void add(long value){

this.count = this.count + value;

}

}

Imagine if two threads, A and B, are executing the add method on the same instance of the Counter class. There is no way to know when the operating system(scheduler) switches between the two threads. The code in the add() method is not executed as a single atomic instruction by the Java virtual machine. Rather it is executed as a set of smaller instructions, similar to this:

Read this.count from memory into PC register.

Add value to PC register.

Write register to memory.

Observe what happens with the following mixed execution of threads A and B:

this.count = 0;

A: Reads this.count into a register (0)

B: Reads this.count into a register (0)

B: Adds value 2 to register

B: Writes register value (2) back to memory. this.count now equals 2

A: Adds value 3 to register

A: Writes register value (3) back to memory. this.count now equals 3

The two threads wanted to add the values 2 and 3 to the counter. Thus the value should have been 5 after the two threads complete execution. However, since the execution of the two threads is interleaved, the result ends up being different.

In the execution sequence example listed above, both threads read the value 0 from memory. Then they add their individual values, 2 and 3, to the value, and write the result back to memory. Instead of 5, the value left in this.count will be the value written by the last thread to write its value. In the above case it is thread A, but it could as well have been thread B.

Race Conditions in Critical Sections

The code in the add() method in the example earlier contains a critical section. When multiple threads execute this critical section, race conditions occur.

More formally, the situation where two threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions. A code section that leads to race conditions is called a critical section.

Preventing Race Conditions

To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

SYNCRONIZATION

Race condition

The situation where two or more threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions.

Critical Section

A code section that leads to race conditions is called a critical section.

eg : Joint Bank Account : shared resource

updateBalance n checkBalance

Preventing Race Conditions

To prevent race conditions from occurring you must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

Synchronization = Applying thread safety or applying locks

When is synchronization required ?

It's required iff multiple threads are sharing SAME common resource(eg : any collection,data file , db table ,socket , reservation...) & one thread is accessing & other one is modifying the resource.

How to apply synchronization in java ?

keyword -- synchronized.

Can appear as method modifier or at block level.

To avoid race condition / guard critical section , you apply synchronization.

Using synchronized keyword , a lock is applied at the object level.(i.e instance of the shared resource

eg : JointAccount)

Important statements

1. lock/monitor can be associated with any java object.

2. When does thrd need to acquire the lock (=enter the monitor)?-- if its invoking either synchronized methods or code from synchronized blocks

3. Can single thrd acquire multiple locks -- YES

4. Blocking trigger

unable to acquire lock(enter monitor) : Blocked on monitor/lock

Un blocked -- lock released / monitor free.(synchronized method rets or synchronized block over)

5. If a thread invokes sleep(or invokes join,yield,notify) or encounters context switching , it holds any locks it has—it doesn't release them.

What's the need of synchronized blocks?

1. Instead of writing long synchronized methods (n thus reducing the performance due to larger extent of the lock) , identify critical section & guard it using synchronized block.

2. While using inherently thread un safe API(StringBuilder, ArrayList,LinkedList,HS,LHS,HM...) in multi thrded environment : you can still apply thread safety : using synchronized blocks.

synchronized block syntax

synchronized(shared resource ref.)

{

Access the methods of shared resource in mutually exclusive manner.

}

1. Only methods (or blocks) can be synchronized, not variables or classes.

2. Each object has just one lock.

3. Not all methods in a class need to be synchronized. A class can have both

synchronized and non-synchronized methods.

4. If two threads are about to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method,only one thread at a time will be able to execute the method. The other thread will need to wait until the first one finishes its method call. In otherwords, once a thread acquires the lock on an object, no other thread can enter ANY of the synchronized methods in that class (for that object).

5. If a class has both synchronized and non-synchronized methods, multiple

threads can still access the class's non-synchronized methods. If you have methods that don't access the data you're trying to protect, then you don't

need to synchronize them. Synchronization can cause a hit in some cases (or

even deadlock if used incorrectly), so you should be careful not to overuse it.

6. If a thread goes to sleep(or invokes join,yield,notify) or encounters context switching , it holds any locks it has—it doesn't release them.

7. A thread can acquire more than one lock. For example, a thread can enter a

synchronized method, thus acquiring a lock, and then immediately invoke

a synchronized method on a different object, thus acquiring that lock as

well. As the stack unwinds, locks are released again. Also, if a thread acquires

a lock and then attempts to call a synchronized method on that same

object, no problem. The JVM knows that this thread already has the lock for

this object, so the thread is free to call other synchronized methods on the

same object, using the lock the thread already has.

eg :

class A {

private B b1;

synchrnized void test()

{

...

b1.testMe();

}

}

class B

{

synchronized void testMe()

{

//some B.L

}

}

Similar can be achieved using nested synchronized blocks.

8. You can synchronize a block of code rather than a method.

When to use synched blocks?

Because synchronization does hurt concurrency, you don't want to synchronize

any more code than is necessary to protect your data. So if the scope of a method is

more than needed, you can reduce the scope of the synchronized part to something

less than a full method—to just a block. OR when u are using Thread un-safe(un-sunchronized eg -- StringBuilder or HashMap or HashSet) classes in your appln.

-----------------------------

Regarding static & non -static synchronized

1. Threads calling non-static synchronized methods in the same class will

only block each other if they're invoked using the same instance. That's

because they each lock on "this" instance, and if they're called using two different

instances, they get two locks, which do not interfere with each other.

2. Threads calling static synchronized methods in the same class will always

block each other—they all lock on the same Class instance.

3. A static synchronized method and a non-static synchronized method

will not block each other, ever. The static method locks on a Class

instance(java.lang.Class<?>) while the non-static method locks on the "this" instance—these actions do not interfere with each other at all