1. 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 behaviours) from the higher hierarchies.
2. 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).
3. By pulling out all the common variables and methods into the super classes, 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.
4. A subclass inherits all the member variables and methods from its super classes (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.
5. A subclass inherits all the variables and methods from its super classes, including its immediate parent as well as all the ancestors.
6. 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{...}

2. multi level inheritance

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

3. multiple inheritance --- NOT supported

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

Why --For simplicity.

Regarding this & super

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

2. Has to be 1st statement in the constructor

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

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

Polymorphism

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

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

    1. Achieved via method overloading
    2. rules -- can be in same class or in sub classes.
    3. same method name
    4. signature -- different (no/type/both)
    5. ret type --- ignored by compiler.

2. Dynamic 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(JVM)

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)
2. scope---must be same or wider.
3. Can not add in its throws clause any new or broader checked exceptions. BUT can add any new unchecked exceptions. Can add any subset or sub-class of checked exceptions.

abstract : keyword in Java

1. abstract methods ---methods only with declaration & no definition.
2. Any time a class has one or multiple abstract methods ---- class must be declared as abstract class.
3. Abstract classes can't be instantiated BUT can create the ref. of abstract class type to refer to concrete sub-class instances.
4. Abstract classes CAN HAVE concrete(non-abstract) methods.
5. Abstract classes MUST provide constructor/s to init its own private data members.

final -- keyword in java

1 final data member(primitive types) - constant.

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

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

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

instanceof -- keyword in java

- used for testing run time type information.
- 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

Interface in Java

What is interface ?

1. An interface in java is a blueprint of a class. It has public static final data members and public n abstract methods only.
2. 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).
3. It is used to achieve full abstraction and multiple inheritance in Java.
4. Java Interface also represents IS-A relationship.
5. 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

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.

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

Exception Handling

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

- JRE(main thread) creates matching Exception class instance(java.io.FileNotFoundException,java.lang.ArrayOutOfBoundsExc, NullPointerExc,ArithmeticExc)
- JRE checks -- if prog has provided Exception handling code ?
    - NO -- JRE aborts java code(by supplying def handler) & prints details --F.Q Exception class name, reason behind failure & location details(err stack trace
    - YES (try---catch) JRE execs Exceptionhandling block & continues with the rest of the code.

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

Inheritance hierarchy of Exception classes

unchecked vs checked Exception.

Creating custom Exception

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

Checked & Unchoked Exception are detected or occur only in run-time.

JRE 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 (exc1 e){} catch (exc2 e) {}catch(Exception e){catch-all}

3.5 3. try {...} catch (exc1 e){...} catch (exc2 | exc3 e) {...}catch(Exception e){catch-all}

4. throws syntax ---

method declaration throws comma separated list of Exception 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.

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

4.5 Throwable class API

1. public String toString() -- rets Name of exc class & reason.

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)

{

}

Creating Custom Exception (User defined exception or application Exception)

Need :

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

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

1. Create a package public class which extends Throwable(not recognize but legal)/Exception(recommended)/Error(not recognize but legal)/RuntimeException (not recognize but legal)

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

throws	throw
keyword in java , for exception handling can appear only in meth declaration. mandatory for un-handled checked excs Meaning --- Current method IS NOT handling the exception BUT its caller should handle. throws' keyword allows delegation of exc handling to the caller.	keyword in java , for exception handling can only appear as java statement Typically used for raising custom exceptions Syntax --- throw Throwable instance; Legal -- throw new NullPointerException("msg"); throw new Emp(.....); ---Illegal Legal -- throw new SocketException("msg",e); Legal --throw new InvalidInputException("invalid email");

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.

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

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.

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

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

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

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

(un checked exc)

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.
- 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 Class:

nner 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.

Regarding wrapper classes

1. What's need of wrapper classes?

    1. to be able to add prim 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. Constructor & 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

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

About Equals :

- 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 equality , based upon prim key criteria(i.e based upon unique ID)

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<>();

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

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

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. 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

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.

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 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.

2. Using PK , override equals for content equality.

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.

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 specifie

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.

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

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

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.

Generic syntax

- Available from Java SE 5 onwards.
- Represents Parameterized Types.(eg : ArrayList<Emp>)
- 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 caught 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> {...}

T ---type --- ref type

T --- only ref types are supported (no primitive types)

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 -- List, 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))

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

Java LinkedList Deque Methods

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 remvoeFirst(): 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.

Difference between ArrayList and LinkedList in Java

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.

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.

HashMap

- 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.
- 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).

JAVA 8 New Features

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

Java 8 enables us to add non-abstract method implementations to interfaces by utilizing 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

In case of ambiguity or to refer to def implement from i/f -- use InterfaceName.super.methodName(...) syntax

2 Can add static methods in java interfaces --- It's a better alternative to writing static library methods in helper class(eg --Arrays or Collections)

Such static methods can't be overridden in implementation class.

BUT can be re-declared.

They have to be invoked using interface name , even in implementation or non implementation classes.(otherwise compiler error)

3. Functional interfaces ---An interface which has exactly single abstract method(SAM) is called functional interface.

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

New annotation introduced -- @FunctionalInterface

(since Java SE 8)

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

4. Lambda Expressions

Regarding functional programming

What is functional programming ?

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

Main concepts of FP are

1. Pure functions & side effects

2. Referential transparency

3. First class functions & higher order functionas.

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

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 behaviour 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 laziness 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).

Lambda expressions

- It's derived from lambda calculus.
- It was a big change in calculus world, which gave tremendous 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}

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.

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.

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.

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)

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

Serialization

Need -- In the absence of Object streams, if u 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.

Persistence ---saving the state of the object on the binary stream.

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 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 object's class is serializable

eg : If Item class HAS - A reference of ShippingAddress

The tree of object's 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 or ref)

transient implies ---skip from serialization.(meant for JVM)

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 --- sub-class developer has to explicitely write the state of super-class.

What happens during deserialization?

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,

2. Class.newInstance(),

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.

Constructor of serializable class does not get called during de-serialization.

What are pre-requisites for de-serialization?

.class file for Class Obj to be de-serialized + Bin data stream containing state.

What is serialversion UID?

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. 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). But you can control this - by adding your own UID.

Serialization format overview

1. Magic no.

2. Serialization format version no.

3. Class desc -- class name,serial version uid,desc of data members(class signature)

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.

What is the need of ObjectInputStream & ObjectOutputStream ?

- To achieve 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

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

A -- transition from rdy to run -----> Running

Triggered by --- in time slice based scheduling --- time slot of earlier thrd over OR in pre-emptive multitasking -- higher prio thrd pre-empts lower prio thrd.

B --- transition from running ---> ready to run

Reverse of earlier transition OR

public static void yield()----

Requests underlying scheduler to swap out current thrd SO THAT some other lower prio thrd MAY get a chance to run. (to avoid thrd starvation -- i.e co-operative multi-tasking.)

C --running state --- Only in this state --- run() method gets executed.

running --->dead --- Triggers -- run() method returns in healthy manner . OR run() aborts due to un-handled , un-checked excs.

D -- blocked ---> rdy to run --- when any of blocking condition is removed --- blocked thrd enters rdy pool & resumes competition among other thrds.

API Involved

1. Thread class constructor to be used in extends Thread scenario

    1.1 Thread() --- A new thrd is created BUT with JVM supplied name.

    1.2 Thread(String nm) -- creates named thread.

2. Thread class constructor to be used in implments scenario

    2.1 Thread (Runnable target/inst) --- Creates a new thrd --- by passing instance of the class which implements Runnable i/f.

Run time significance -- Whenever this thrd gets a chance to run --- underlying task scheduler -- will invoke(via JVM) this class's run() method.

2.2 Thread (Runnable inst,String name)

Thread class API

1.public String getName()

2. public void setName(String nm)

3. public static Thread currentThread() -- rets ref of the invoker thrd.

4. public int getPriority() -- rets current prio.

Prio scale -- 1---10(MIN_PRIORITY,MAX_PRIORITY)

NORM_PRIORITY ---- 5

4.5 public void setPriority(int prio) --- must be invoked before start()

DO NOT rely on priority factor -- since it is ultimately specific to underlying OS

5. public String toString() --- to ret -- name,prio & thrd grp name

6. public static void sleep(long ms) throws InterruptedException

When is synchronization(=applying thread safety=locking shared resource) required?

In multi-threaded java applns -- iff multiple thrds trying to access SAME copy of the shared resource(eg -- reservation tkt,db table,file or socket or any shared device) & some of the threads are reading n others updating the resource

How to lock the resource?

Using synchronized methods or synchronized blocks.

In either approach : the java code is executed from within the monitor & thus protects the concurrent access.

Note : sleeping thrd sleeps inside the monitor(i.e Thread invoking sleep(...) , DOESn't release the ownership of the monitor)

eg classes :

StringBuilder : thrd-unsafe.--- unsynchronized --- if multiple thrds try to access the same copy of the SB, SB may fail(wrong data)

StringBuffer --- thrd -safe ----synchronized internally--- if multiple thrds try to access the same copy of the SB, only 1 thrd can access the SB at any parti. instance.

which is reco class in single threaded appln? --- StringBuilder

multiple thrds -- having individual copies -- StingBuilder

multiple thrds -- sharing same copy -- StringBuilder --

identify code to be guarded -- sb 's api -- invoke thrd unsafe API -- from inside synched block.

ArrayList(inherently thrd un-safe) Vs Vector(inherently thrd safe)

HashMap(un-safe) Vs Hashtable(thrd safe)

synchronized block syntax --- to apply synchro. externally.

synchronized (Object to be locked--- shared resource)

{

//code to be synchronized --methods of shared res. -- thrd safe manner(from within monitor)

}

1. If any thrd is accessing any synched method of 1 obj, then same thrd or any other thrd CANT concurrently access same method of the same obj.(Tester1.java)

2. If any thrd is accessing any synched method of 1 obj, then same thrd or any other thrd CANT concurrently access same method or any other synchronized method of the same obj.(Tester2.java)

2. If thrds have their own independent copies of resources, synch IS NOT required.(Tester2.java)

3.If u are using any thrd un-safe code(ie. ready code without source) --& want to apply thrd safety externally --- then just wrap the code within synched block to use locking.(Tester3.java)

Objective : Create Producer & Consumer thrds .

Producer produces data samples & consumer reads the same.

For simplicity : let the data be represented by : single Emp record

Producer produces emp rec sequentially & consumer reads the same.

Rules : 1 when producer is producing data , consumer thrd concurrently should not be allowed to read data & vice versa.

Any more rules?????????????

Yes --- correct sequencing is also necessary in such cases.

Rule 2 : Producer must 1st produce data sample ---consumer reads data sample & then producer can produce next data sample. Similarly consumer should not be able to read stale(same) data samples .

ITC --- API level

Object class API

1. public void wait() throws IE ---thrd MUST be owner of the monitor(i.e invoke wait/notify/notifyAll from within synched block or method) --- othewise MAY get IllegalMontitorStateExc

---causes blocking of the thrd outside montitor.

UnBlocking triggers --- interrupt(not reco --- since it may cause death of thrd) , notify/notifyAll --- reco.

2. 1. public void wait(long ms) throws IE

UnBlocking triggers --- interrupt(not reco --- since it may cause death of thrd) , notify/notifyAll --- reco.,tmout exceeded

2.2 public void notify() -- MUST be invoked from within monitor , ow may get IllegalMonitorStateExc

Un-blocks ANY waiting thread , blocked on SAME MONITOR

2.3 public void notifyAll() -- Un-blocks ALL waiting threads , blocked on SAME MONITOR

notify/notifyAll--- DOESN't BLOCK the thread & Doesn't release lock on monitor. --- send wake up signal -- to thrd/s waiting on same monitor.

wait --- Blocks the thread --- Releases lock on the monitor.

volatile --- java keyword, applicable at data member.

typically used in multi-threaded scenario only when multiple thrds are accessing the same data member.

Use --- to specify-- that data var. is being used by multiple thrds concurrently -- so dont apply any optimizations(OR the value of the variable can get modified outside the current thrd) . With volatile keyword -- its guaranteed to give most recent value.

The volatile modifier tells the JVM that a thread accessing the variable must always get its own private copy of the variable with the main copy in memory

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

9. public void join() throws InterruptedException

Blocking method(API)

--Causes the invoker thread to block till specified thread gets over.

join method can be used effectively to avoid orphan threads

10. public void join(long ms) throws InterruptedException

--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

Regarding 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.

Regarding synchronization

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 other words, 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.

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-synchronized eg -- StringBuilder or HashMap or HashSet) classes in your application.

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.