When you create String object using new() operator, it always create a new object in [heap memory](http://java67.blogspot.sg/2013/08/guide-of-javalangoutofmemoryerror-java-heap-space-tomcat-eclipse-minecraft-jboss.html). On the other hand, if you create object using String literal syntax e.g. "Java", it may return an existing object from String pool..

String a = "Java";

String b = "Java";

System.out.println(a == b); // True

Here two different objects are created and they have different references:

String c = new String("Java");

String d = new String("Java");

System.out.println(c == d); // False

**Questions about Final Keyword:**

**Final** Keyword should be **initialized**, same time when declared.

What is the use of final keyword in Java?  
**(A)** When a class is made final, a sublcass of it can not be created. – cannot be extended.  
**(B)** When a method is final, it can not be overridden.  
**(C)** When a variable is final, it can be assigned value only once.

Blank Final variable && Static final variable -> Note: (cannot initialized by any method call)

public class Employee

{

//One way to do so:

  final static double salary;

  static           // static block

  {

    salary = 9999.99;

  }

// Second way to do so: through constructor

final double salary;

  Employee() //via constructor

  {

    salary = 9999.99;

  }

}

* **Interface:** if we make it **final** it cannot be implemented. Only **public & abstract** are permitted while creating an interface.
* **Constructor**: **No,** Constructor cannot be declared as **final**. Constructors are not inherited and so it cannot be overridden, so there is no use to have a final constructor.
* **Within Method**:

public static void main(String args[])

{

// Local final variable

final int SPEED;

SPEED = 25;

}

* **Yes,** the final method can be [**overloaded**](https://www.javainterviewpoint.com/java-method-overloading-example/) but cannot be**overridden**. Which means you can have more than one final method with the same name with different parameters.
* **OOP Related Questions: (**Inheritance, polymorphism, Abstraction, Encapsulation, Composition, Association, Aggregation)

OOP is to write programs based on the real world objects. The states and behaviors of an object are represented as the member variables and methods.

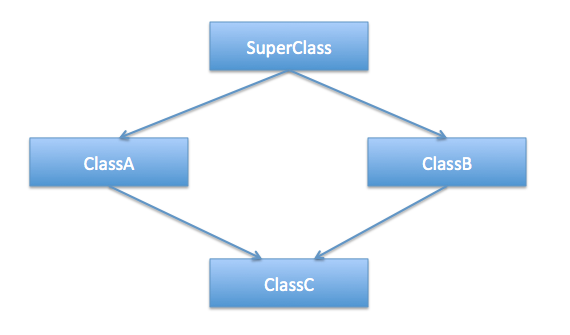
* **Class:** It is the blueprint that describes the details of an object. Type of Object.
* An object is an instance of a class. It has its own state, behavior, and identity.
* Abstraction:  construct the structure of the objects.
* Encapsulation: to create and define the permissions and restrictions . As example:  to make the member variables of a class private and providing public getter and setter methods- POJO class

That hidden data can be restricted to the members of that class.

* Polymorphism: Polymorphism is the occurrence of something in various forms. Java supports various forms of polymorphism like polymorphic reference variables, polymorphic method, polymorphic return types and polymorphic argument types.
* Inheritance: A subclass can inherit the states and behaviors of it’s super class is known as inheritance.

**Diamond problem in Inheritance**: In case of multiple inheritance, suppose class A has two subclasses B and C, and a class D has two super classes B and C.If a method present in A is overridden by both B and C but not by D then from which class D will inherit that method B or C? This problem is known as diamond problem.

**Why Java does not support multiple inheritance?** -- Java was designed to be a simple language and multiple inheritanceintroduces complexities like **diamond problem**.



**Note: Multiple inheritance is achived by intefaces in java**

/\*

\* This is perfectly fine because the interfaces are only declaring the methods

\* and the actual implementation will be done by concrete classes implementing

\* the interfaces. So there is no possibility of any kind of ambiguity in

\* multiple inheritances in Java interfaces.

\*/

**interface** InterfaceA {

**public** **void** doSomething();

}

**interface** InterfaceB {

**public** **void** doSomething();

}

**interface** InterfaceC **extends** InterfaceA, InterfaceB {

//same method is declared in InterfaceA and InterfaceB both

**public** **void** doSomething();

}

\*\*\*\* This way a class can have multiple inheritance ->

**class** InterfacesImpl **implements** InterfaceA, InterfaceB, InterfaceC {

@Override

**public** **void** doSomething() {

System.***out***.println("doSomething implementation of concrete class");

}

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

InterfaceA objA = **new** InterfacesImpl();

InterfaceB objB = **new** InterfacesImpl();

InterfaceC objC = **new** InterfacesImpl();

//all the method calls below are going to same concrete implementation

objA.doSomething();

objB.doSomething();

objC.doSomething();

}

}

* **Abstraction Questions:-> Interface and Abstract class**

**Can abstract class have constructors in Java?**

Yes, abstract class can declare and define constructor in Java  
Since you can not create instance of abstract class,  constructor can only be called during constructor chaining, to initialize common variables,

##Yes, abstract class can implement interface by using implements keyword , Note it: Since they are abstract, they don’t need to implement all methods.

**No, abstract class cannot be final in Java. Making them final will stop abstract class from being extended, which is the only way to use abstract class**  
Can abstract class have static methods in Java?

Yes, abstract class can declare and define static methods, nothing prevents from doing that. But, you must follow guidelines for making a method static in Java, as it’s not welcomed in a object oriented design, because static methods can not be overridden in Java. It’s very rare, you see static methods inside abstract class, but as I said, if you have very good reason of doing it, then nothing stops you.

**Can you create instance of abstract class?**

No, you can not create instance of abstract class in Java, they are incomplete. Even though, if your abstract class don’t contain any abstract method, you can not create instance of it. By making a class abstract

**Is it necessary for abstract class to have abstract method?**

No, It’s not mandatory for an abstract class to have any abstract method.

Note: Interface can have Definition in method using STATIC or DEFAULT.

**public** **interface** LamdaInterface {

**void** printIt(String text);

**default** **public** **void** printUtf8To(String text, OutputStream outputStream){

System.***out***.println("You can have a interface with defination "

+ "using STATIC AND DEFAULT");

}

**static** **void** printItToSystemOut(String text){

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

}

}

**Difference between abstract class and interface in Java?**

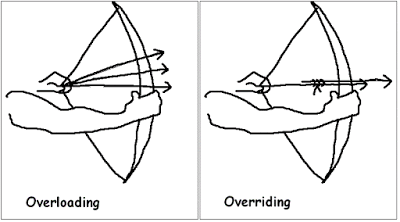
**Abstract class vs Interface**

1. **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
2. **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
3. **Type of variables:**Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
4. **Implementation:** Abstract class can provide the implementation of interface. Interface can’t provide the implementation of abstract class.
5. **Inheritance vs Abstraction:** A Java interface can be implemented using keyword “implements” and abstract class can be extended using keyword “extends”.
6. **Multiple implementation:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
7. **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc..

**When do you favor abstract class over interface?**

An abstract method is a method without body. You just declare method, without defining it and use abstract keyword in method declaration.  All method declared inside Java Interface are by default abstract

--Yes, abstract class can contain main method, it just another static method and you can execute Abstract class with main method, until you don’t create any instance.

* **Overloading and overriding Questions:**
* overloading is a process of declaring two methods with same name but different method signature
* Method overriding is another way to define method with same name but different code but it must be in sub class. Overriding is based upon run-time Polymorphism
* **What is method hiding in Java?**  
  static method cannot be overriding in Java because their method calls are resolved at compile time but it didn't prevent you from declaring method with same name in sub class. In this case we say that method in sub class has hided static method from parent class
* Fundamental difference between overloading and overriding is that former took place during compile time while later took place during run-time
* private, static and final method cannot be overridden in Java.
* 
* In a subclass (or Derived Class), we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.
* **What is method signature?**
* **A) Method name**
* **b) Number of arguments**
* **c) Types of arguments**

**Note: Simple, when method signature is different then there is compilation error, No other things matter for overriding….**

**Is it possible to have two methods in a class with same method signature but different return types?-**

No, compiler will give duplicate method error. Compiler checks only method signature

**Can we overload main() method?**

Yes, we can overload main() method. A class can have any number of main() methods but execution starts from **public static void main(String[] args)** only.

* Why Static method cannot be overridden?

Because if you call childMethod it always print parentMethod so this case does not follow overloading. This case is also called **Method Hiding.**

**class** TestChild **extends** TestParent{

**public** **static** **void** form() {

System.***out***.println("Child form");

}

}

**class** TestParent {

**public** **static** **void** form() {

System.***out***.println("Parent form");

}

}

//Driver Class//

**public** **class** TestClass {

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

TestParent t1 = **new** TestParent();

TestParent t2 = **new** TestChild();

TestChild t3 = **new** TestChild();

t1.*form*(); -------**OutPut**:- Parent form

t2.*form*(); -------**OutPut**:- Parent form

t3.*form*(); -------**OutPut**:- Child form

}

}

#### What is static binding?

When you compile Java program. During compilation process, compiler bind method call to actual method. This is called static binding and method overloading binding happens at compile time.

#### What is dynamic binding?

Binding of overridden methods happen at runtime is known as dynamic binding.

Note: Overloaded Method is bounded by Static binding while Overridden Method are bounded by Dynamic bounding in java.

An **Is-A relationship** is also known as inheritance and a **Has-A relationship** is also known as composition in Java.

When there is an extends or implement keyword in the class declaration in Java, then the specific class is said to be following the Is-A relationship.

What is the meaning of “IS-A” and “HAS-A” relationship?

“IS-A” relationship implies inheritance. A sub class object is said to have “IS-A” relationship with the super class or interface. If class A extends B then A “IS-A” B. It is transitive, that is, if class A extends B and class B extends C then A “IS-A” C. The “**instanceof**” operator in java determines the “IS-A” relationship.

When a class A has a member reference variable of type B then A “HAS-A” B. It is also known as Aggregation.

Eg: Student “Has-A” Address. i.e – student has address reference as member. Both are connected via reference while in “IS-A” both are connected with parent-child concept.

**class** Student{

**int** id = 1;

Address add;

}

**class** Address {

String city = “Banglore”;

}

**Assosiation** (Has-A relationship): has two type- **Composition and Aggression.**

**Composition** is two way traffic- interdependent. Books and library is interdependable- so if library destroyed all books in library destroyed.

**class** Book {

**public** String title;

**public** String author;

Book(String title, String author) {

**this**.title = title;

**this**.author = author;

}

}

// Libary class contains

// list of books.

**class** Library {

// reference to refer to list of books.

**private** **final** List<Book> books;

Library(List<Book> books) {

**this**.books = books;

}

**public** List<Book> getTotalBooksInLibrary() {

**return** books;

}

}

**Aggression** is one way traffic. – Address is still existing when student is removed.

**class** Student{

**int** id = 1;

Address add;

}

**class** Address {

String city = “Banglore”;

}

**Aggregation vs Composition**

1. **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don’t exist separate to a Human
2. **Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”** relation.
3. **Type of association:**Composition is a **strong** Association whereas Aggregation is a **weak** Association.

* **InstanceOf In java:**

When we do typecast, it is always a good idea to check if the typecasting is valid or not. instanceof helps us here. We can always first check for validity using instancef, then do typecasting.

**class** Parent

{

**int** value = 1000;

}

**class** Child **extends** Parent

{

**int** value = 10;

}

// Driver class

**class** Test

{

**public** **static** **void** main(String[] args)

{

Parent cobj = **new** Child();

Parent par = cobj;

// Using instanceof to make sure that par

// is a valid reference before typecasting

**if** (par **instanceof** Child)

{

System.***out***.println("Value accessed through " +

"parent reference with typecasting is " +

((Child)par).value);

}

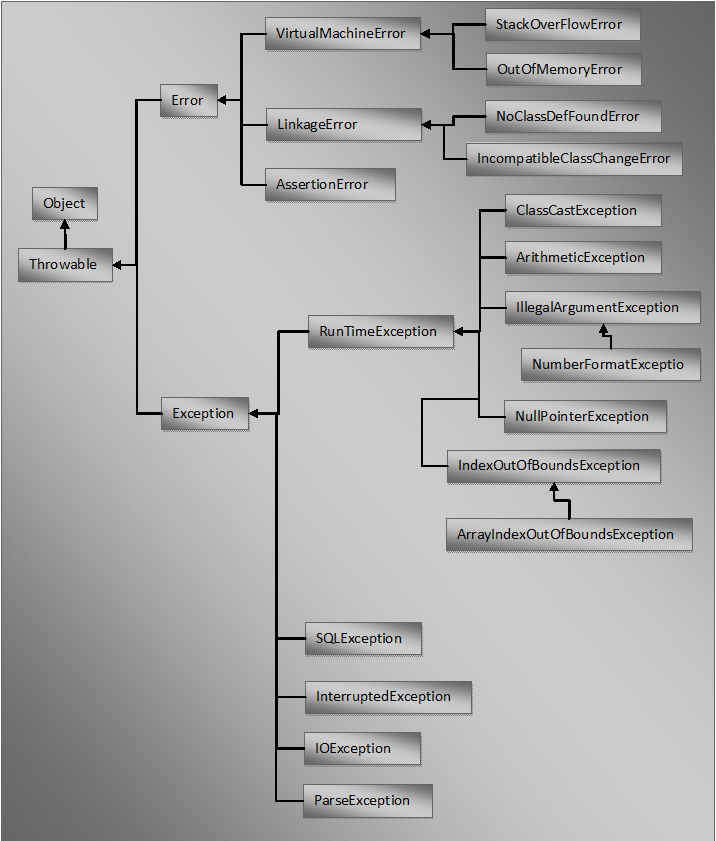
}

}

**Access Modifiers in Java**: (Least Accessible) **private** < **default** < **protected** < **public** (Most Accessible)



**Exception:**



**Constructor Questions:**

**Constructor** will be invoked during the time of object creation. It initializes all the class verriables by default values.

Note:  first statement in any constructor implicitly calls super(), default super class constructor in Java.

1. Constructor **name** should be the same as the class name
2. It **cannot** contain any **return type**
3. It **can** have all **Access Modifiers** are allowed (private , public, protected, default)
4. It **Cannot** have any **Non Access Modifiers** (final ,static, abstract, synchronized)
5. It **can** take any number of **parameters**
6. Constructor can **throw exception**, we can have**throws clause**

* **No**, We cannot have a Constructor defined in an **Interface**.
* **Private constructor is used in singleton class.**
* **Constructor Chaining** is nothing but calling one Constructor from another. **this keyword** is used to call the **current** class constructor and **super keyword** is used to call the **parent** class constructor.
* public class Child extends Parent
* {
* public Child()
* {
* **this("JIP");**
* System.out.println("Child class no-args constructor called");
* }
* public Child(String name)
* {
* **super("JIP");**
* System.out.println("Child class Parameterized constructor called by "+name);
* }
* **No,** we **cannot** have **this** and **super** in a same constructor as anyone only can be in the first line of the constructor.
* **No**. You cannot call a sub class constructor from a super class constructor.
* **Yes,** an abstract class can have a constructor. The below code work perfectly fine.
* A constructor is not inherited, so there is no need for declaring it as **final**.
* When you set a method as **static**, it means “The Method belong to class and not to any particular object” but a **constructor** is always invoked with respect to an object, so it makes no sense for a constructor to be **static**.
* **Private constructor** is used in **Singleton class:** Below is the prerequisites.
* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

**class** Singleton {

// static variable single\_instance of type Singleton

**private** **static** Singleton *single\_instance* = **null**;

// private constructor restricted to this class itself

**private** Singleton() {

String s = "Hello I am a string part of Singleton class";

}

// static method to create instance of Singleton class

**public** **static** Singleton getInstance() {

**if** (*single\_instance* == **null**)

*single\_instance* = **new** Singleton();

**return** *single\_instance*;

}

}

//Driver Method

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

// instantiating Singleton class with variable x

Singleton x = Singleton.*getInstance*();

}

* There are many classes in JDK which is implemented using Singleton pattern like java.lang.Runtime which provides getRuntime()
* How do you prevent for creating another instance of Singleton using clone() method?  
    
  -by Clonning and by refrlection.

**Deep Copy And Shallow Copy:**

Deep copy copies also inner objects of the object wj=hile shallo copy only copies the main object so inner objects will be of same of original therefore there is chances that if copied object can modify the inner objects of the original object. See the example of both:

Shallow copy:->

public class Person {

private Name name;

private Address address;

public Person(Person originalPerson) {

this.name = originalPerson.name;

this.address = originalPerson.address;

}

[…]

}

Deep copy:->

public class Person {

private Name name;

private Address address;

public Person(Person otherPerson) {

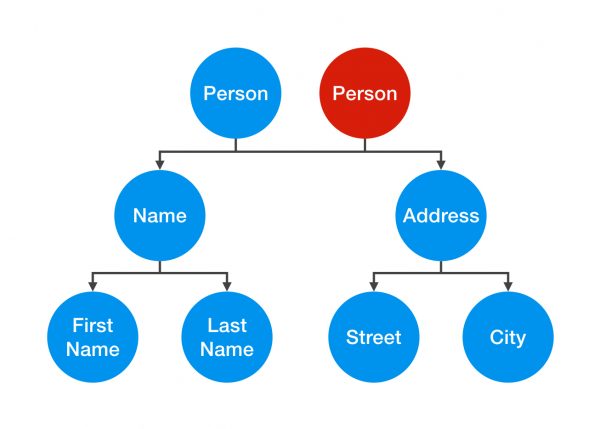
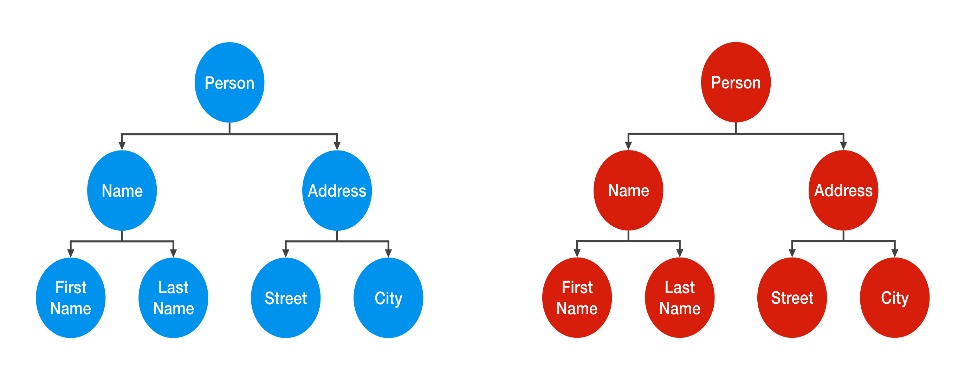
this.name = new Name(otherPerson.name);

this.address = new Address(otherPerson.address);

}

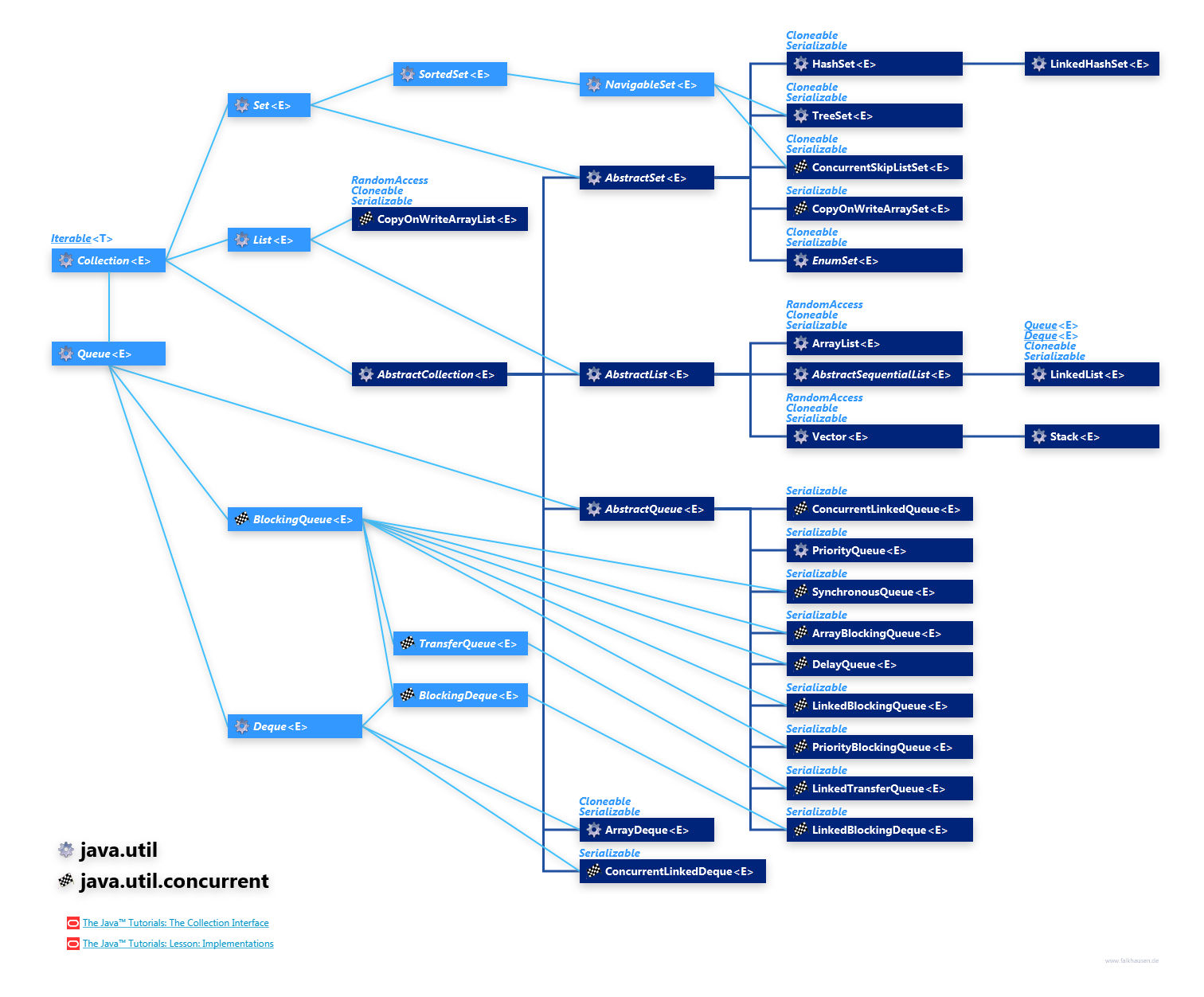
[…]

}

**Collection Questions-**

**Collection Hirarchy-**



-A collection is **an object that represents a group of objects.**

**Signature of Collection interface:-**

public interface Collection extends Iterable { -////- }

Why Map interface does not extend Collection interface?

A good answer to this interview question is “**because they are incompatible**“. Collection has a method add(Object o). Map can not have such method because it need key-value pair. There are other reasons also such as Map supports keySet, valueSet etc. Collection classes does not have such views.

-- Collections.reverse(list);

#### How HashSet store elements?

You must know that HashMap store key-value pairs, with one condition i.e. keys will be unique. HashSet uses Map’s this feature to ensure uniqueness of elements. In HashSet class, a map declaration is as below:

|  |
| --- |
| private transient HashMap<E,Object> map;    //This is added as value for each key  private static final Object PRESENT = new Object(); |

So **when you store a element in HashSet, it stores the element as key in map and “PRESENT” object as value**. (See declaration above).

|  |
| --- |
| public boolean add(E e) {  return map.put(e, PRESENT)==null;  } |

# hashmap, largely depends on hashCode() and equals() method of Key objects. So a good key object**must provide same hashCode() again and again**, no matter how many times it is fetched.

# Map interface provides 3 views of key-values pairs stored in it:

* key set view
* value set view
* entry set view

All the views can be navigated using iterators.

#Diffence between hashmap and treemap.

TreeMap is special form of HashMap. **It maintains the ordering of keys** which is missing in HashMap class. This ordering is **by default “natural ordering”**. Treemap uses compateTo method to desired ordering.

* Difference between set and List:

Set is unordered collection where List is ordered collection based on zero based index.

List allow duplicate elements but Set does not allow duplicates.

List does not prevent inserting null elements (as many you like), but Set will allow only one null element.

* Diffrence between Hashtable and HashMap:
* Hashtable is synchronized, whereas HashMap is not.
* Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
* The third significant difference between HashMap vs Hashtable is that Iterator in the HashMap is a fail-fast iterator while the **enumerator** for the Hashtable is not.
* **Differnce between Iterator and Enumaration:**
* Iterators allow the caller to remove elements from the underlying collection during the iteration with its remove() method. You can not add/remove elements from a collection when using enumerator.
* Enumeration is available in legacy classes i.e Vector/Stack etc. whereas Iterator is available in all modern collection classes.
* Differnce between Iterator ans Listiterator:
* By using Iterator we can retrieve the elements from Collection Object in forward direction only whereas List Iterator, which allows you to traverse in either directions using hasPrevious() and previous() methods.
* ListIterator allows you modify the list using add() remove() methods. Using Iterator you can not add, only remove the elements.
* **Difference between TreeSet and SortedSet?**

SortedSet is an interface which TreeSet implements. That’ it !!

* **Difference between ArrayList and LinkedList?**
* LinkedList store elements within a doubly-linked list data structure. ArrayList store elements within a dynamically resizing array.
* you can walk the list forwards or backwards, but grabbing an element in the middle takes time proportional to the size of the list. ArrayLists, on the other hand, allow random access, so you can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap.
* LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

**How to make a collection read only?**

While passing a Collection as argument to a function, how can we make sure the function will not be able to modify it? -- We can create a read-only collection using Collections.unmodifiableCollection(Collection c) method before passing it as argument, this will make sure that any operation to change the collection will throw UnsupportedOperationException.

Use following methods:

* Collections.unmodifiableList(list);
* Collections.unmodifiableSet(set);
* Collections.unmodifiableMap(map);

**How to make a collection thread safe?**

Use below methods:

* Collections.synchronizedList(list);
* Collections.synchronizedSet(set);
* Collections.synchronizedMap(map);

**What do you understand by iterator fail-fast property?**

**Fail-fast Iterators fail as soon as they realized that structure of Collection has been changed since iteration has begun.** Structural changes means adding, removing or updating any element from collection **while one thread is Iterating over that collection.**

**Fail-safe iterators** are just opposite to fail-fast. **They never fail if you modify the underlying collection on which they are iterating**, because they work on clone of Collection instead of original collection and that’s why they are called as fail-safe iterator. Iterator of **CopyOnWriteArrayList** is an example of fail-safe Iterator , and never throw ConcurrentModificationException

**Which collection classes provide random access of it’s elements?**

ArrayList, HashMap, TreeMap, Hashtable classes provide random access to it’s elements.

**What is Comparable and Comparator interface?**

**Comparable**

A comparable object is capable of comparing itself with another object. The class itself must implement the java.lang.Comparable interface in order to be able to compare its instances. 

**Comparator**

A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class’s instances. This comparator class must implement the java.util.Comparator interface.

These needs to be implemented by user. Those are;  
**java.lang.Comparable: int compareTo(Object o1)**  
This method compares this object with o1 object. Returned int value has the following meanings. 

1. positive – this object is greater than o1
2. zero – this object equals to o1
3. negative – this object is less than o1

**java.util.Comparator: int compare(Object o1, Objecto2)**  
This method compares o1 and o2 objects. Returned int value has the following meanings. 

1. positive – o1 is greater than o2
2. zero – o1 equals to o2
3. negative – o1 is less than o2
4. **java.util.Collections.sort(List, Comparator)** and **java.util.Arrays.sort(Object[], Comparator)**methods can be used if a Comparator is available for comparison.

Below Exmple best explained comparator and comparable->

**import** java.util.\*;

**public** **class** TestClass {

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

List coll = Util.*getEmplyees*();

// Comparable sort method. In this case we need to implement

//comparable method in employee class.

Collections.sort(coll); // Comparable sort method

// In this case we need to implement comparable method in employee class. We //should remove employee class comparable. For every attribute sorting we should //write a class implementing comparator like below class EmpSortByName()

Collections.*sort*(coll, **new** EmpSortByName());// comparator sort method

*printList*(coll);

}

**private** **static** **void** printList(List<Employee> list) {

System.***out***.println("EmpId\tName\tAge");

**for** (Employee e : list) {

System.***out***.println(e.getEmpId() + "\t" + e.getName() + "\t" + e.getAge());

}

}

}

**class** Employee **implements** Comparable<Employee> {

**private** **int** empId;

**private** String name;

**private** **int** age;

@Override

**public** **int** compareTo(Employee o) {

**return** **this**.empId - o.empId;

}

//getter setters

**public** **int** getEmpId() {

**return** empId;

}

**public** String getName() {

**return** name;

}

**public** **int** getAge() {

**return** age;

}

**public** Employee(**int** empId, String name, **int** age) {

**this**.empId = empId;

**this**.name = name;

**this**.age = age;

}

}

// Comparator class which will sort the collection by name

**class** EmpSortByName **implements** Comparator<Employee>{

**public** **int** compare(Employee o1, Employee o2) {

**return** o1.getName().compareTo(o2.getName());

}

}

// Comparator class which will sort the collection by EmpId

**class** EmpSortByEmpId **implements** Comparator<Employee>{

**public** **int** compare(Employee o1, Employee o2) {

**return** o1.getEmpId() - o2.getEmpId();

}

}

**class** Util {

**public** **static** List<Employee> getEmplyees() {

List<Employee> col = **new** ArrayList();

col.add(**new** Employee(5, "Frank", 28));

col.add(**new** Employee(1, "Jorge", 19));

col.add(**new** Employee(6, "Bill", 34));

col.add(**new** Employee(3, "Michel", 10));

col.add(**new** Employee(7, "Simpson", 8));

col.add(**new** Employee(4, "Clerk", 16));

col.add(**new** Employee(8, "Lee", 40));

col.add(**new** Employee(2, "Mark", 30));

**return** col;

}

}

**Questions About ArrayList:**

* Java Array List is the resizable array implementation of list interface.
* Big-O Complexity Access Θ(1) SearchΘ(n) InsertionΘ(n) DeletionΘ(n)
* Java ArrayList is not synchronized.

**Custom ArrayList:**

**package** javaInterviewProgram;

**import** java.util.Arrays;

**public** **class** CustomArrayList {

**private** Object[] arrayPlaceHolder;

**private** **int** actSize = 0;

**public** CustomArrayList() {

arrayPlaceHolder = **new** Object[10];

}

**public** Object get(**int** index) {

**if** (index < actSize) {

**return** arrayPlaceHolder[index];

} **else** {

**throw** **new** ArrayIndexOutOfBoundsException();

}

}

**public** **void** add(Object obj) {

**if** (arrayPlaceHolder.length - actSize <= 5) {

increaseListSize();

}

arrayPlaceHolder[actSize++] = obj;

}

**public** Object remove(**int** index) {

**if** (index < actSize) {

Object obj = arrayPlaceHolder[index];

arrayPlaceHolder[index] = **null**;

**int** tmp = index;

**while** (tmp < actSize) {

arrayPlaceHolder[tmp] = arrayPlaceHolder[tmp + 1];

arrayPlaceHolder[tmp + 1] = **null**;

tmp++;

}

actSize--;

**return** obj;

} **else** {

**throw** **new** ArrayIndexOutOfBoundsException();

}

}

**public** **int** size() {

**return** actSize;

}

**private** **void** increaseListSize() {

arrayPlaceHolder = Arrays.*copyOf*(arrayPlaceHolder, arrayPlaceHolder.length \* 2);

System.***out***.println("nNew length: " + arrayPlaceHolder.length);

}

// Driver Code below

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

CustomArrayList customArrayList = **new** CustomArrayList();

customArrayList.add(**new** Integer(1));

customArrayList.add(**new** Integer(2));

customArrayList.add(**new** Integer(3));

customArrayList.add(**new** Integer(4));

customArrayList.add(**new** Integer(5));

**for**(**int** i=0;i<customArrayList.size();i++){

System.***out***.print(customArrayList.get(i)+" ");

}

customArrayList.add(**new** Integer(9));

System.***out***.println("Element at Index 3:"+customArrayList.get(3));

System.***out***.println("Size: "+customArrayList.size());

System.***out***.println("Removing element at index 1: "+customArrayList.remove(1));

**for**(**int** i=0;i<customArrayList.size();i++){

System.***out***.print(customArrayList.get(i)+" ");

}

}

}

How to synchronize Array List in Java?

Using thread-safe variant of ArrayList which is :**CopyOnWriteArrayList**

Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException

**How to remove duplicates in Arraylist?**

By copy arraylist in set and then copy again in arraylist.

Let suppose Arraylist arraylist;

Set setCoppied = new set(arraylist);

arraylist.clear();

arraylist.addAll(setCoppied);

Now arraylist will have no duplicate element.

* **To Reverse the arraylist:**

Collections.reverse(arraylist);

\* Arraylist initializes with an initial capacity of ten.

**To convert Arraylist into Array**

Object[] array = list.toArray();

        //Printing the elements of the returned array.

        for (Object object : array)

        {

            System.out.println(object);

        }

* **LINKEDLIST Interview Questions:**
* **-** Java LinkedList is the Doubly-linked list implementation of the list interface
* LinkedList can be iterated in reverse direction using descendingIterator()
* LinkedList maintains insertion order.
* LinkedList manipulation is fast because no shifting needs to be occurred.

- Each unit or element of the list is referred as a node. Each node has its own data and the address of the next node

-Singly Linked list are a type of data structure.  In a singly linked list, each node in the list stores the contents of the node and a reference or pointer to the next node in the list.  It does not store any reference or pointer to the previous node.

|  |  |
| --- | --- |
| * Arraylist-We can reach there directly if we have to go to a particular element | * LinkedList-To reach a particular node, you need to go through all those nodes that come before that node. |

* In Linked Lists you don’t need to know the size in advance.
* Linked lists let you insert elements at the beginning and end of the list.

-The biggest benefit of linked lists is that you do not specify a fixed size for your list. The more elements you add to the chain, the bigger the chain gets.

-Strength :Adding elements at either end of a linked list is O(1)*.* Removing the first element is also O(1)*O*(1).

-Weekness: Costly lookups. To access or edit an item in a linked list, you have to take O(i) time to walk from the head of the list to the i th item.

Code Written in Java source file for Node in linkedList.java

**private** **static** **class** Node<E> {

E item;

Node<E> next;

Node<E> prev;

Node(Node<E> prev, E element, Node<E> next) {

**this**.item = element;

**this**.next = next;

**this**.prev = prev;

}

}

Add() Method in LinkedList:

**public** **void** addFirst(E e) {

linkFirst(e);

}

**private** **void** linkFirst(E e) {

**final** Node<E> f = first;

**final** Node<E> newNode = **new** Node<>(**null**, e, f);

first = newNode;

**if** (f == **null**)

last = newNode;

**else**

f.prev = newNode;

size++;

modCount++;

}

Worst Case

space O(n)

prepend O(1)

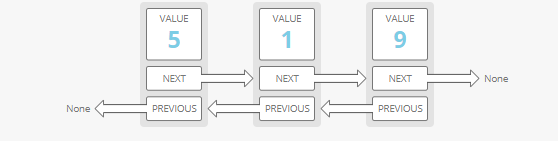
append O(1)

lookup O(n)

insert O(n)

delete O(n)

**Doubly linked list:**



if you just had a pointer to a node in the middle of a list, there would be no way to know what nodes came before it. Not a problem in a doubly linked list.

## **How to find if linked list has a loop?**

***Ananswered***

**Set Interview Question:**

Set does not have duplicate elements because set internally works/implemented upon map(key).

**public** **static** **void** **main**(String[] args)

{

// TODO Auto-generated method stub

HashSet<Object> hashset = **new** HashSet<Object>();

hashset.add(**3**);

hashset.add("Java Hungry");

hashset.add("Blogspot");

hashset.add("Blogspot");

System.out.println("Set is "+hashset);

}

}

*It will print the result* :       Set is [3, Java Hungry, Blogspot]

**Internal implementation of set – add method(where it uses map & put)**

**public** **class** **HashSet**<E>

**extends** AbstractSet<E>

**implements** Set<E>, Cloneable, java.io.Serializable

{

**private** **transient** HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map

**private** **static** **final** Object PRESENT = **new** Object();

**public** **HashSet**() {

*map =* ***new*** *HashMap<>();*

}

// SOME CODE ,i.e Other methods in Hash Set

**public** **boolean** **add**(E e) {

**return** map.put(e, PRESENT)==**null**;

}

// SOME CODE ,i.e Other methods in Hash Set

}

The main point to notice in above code is that put (key,value) will return  
1.  null , if key is unique and added to the map  
2.  Old Value of the key , if key is duplicate

Same as Map:  
Default Initial Capacity of HashSet Object : 16  
Initial Load Factor of HashSet Object : 0.75

**Why Java do not have ConcurrentHashSet class just like ConcurrentHashMap ?**

There is no need to have ConcurrentHashSet class in Java . The reason is you can produce a ConcurrentHashSet  backed by ConcurrentHashMap by using newSetFromMap method.

***Lambda Expressions:*** <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

Lambda Expressions syntax is **(argument) -> (body)**.

Based on functional interface. – in java.util.funnction package

Basic lambda Expression:

(x, y) -> x + y  //This function takes two parameters

                    //and return their sum.

## Lambda Parameters:

Since Java lambda expressions are effectively just methods, lambda expressions can take parameters just like methods. These parameters have to match the parameters of the method on the single method interface. In this case, these parameters have to match the parameters of the onStateChange() method of the StateChangeListener interface

When a lambda expression takes a single parameter, you can also omit the parentheses, like this:

param -> System.out.println("One parameter: " + param);

**var Parameter Types from Java 11**

From Java 11 you can use the var keyword as parameter type. The var keyword was introduced in Java 10 as **local variable type inference**. From Java 11 var can also be used for lambda parameter types. Here is an example of using the Java var keyword as parameter types in a lambda expression:

Function<String, String> toLowerCase = (var input) -> input.toLowerCase();

You can shorten the code this way:

(a1, a2) -> { return a1 > a2; }

You can write:

(a1, a2) -> a1 > a2;

**interface** CheckPerson {

**boolean** test(Person p);

}

Note: you can use the Predicate<T> interface(Predefined in java.util.function) in place of CheckPerson. This interface contains the method boolean test(T t):

interface Predicate<T> {

boolean test(T t);

}

--

new Thread(

            () ->   {

                        System.out.println("My Runnable");

                    }

         ).start();

--

## Java 8 lambda expression examples

I am listing out some code samples which you can read and analyze to how a lambda expression can be used in day to day programming.

**1) Iterating over a List and perform some operations**

|  |
| --- |
| List<String> pointList = new ArrayList();  pointList.add("1");  pointList.add("2");    pointList.forEach(p ->  {                              System.out.println(p);                              //Do more work                          }                   ); |

**2) Create a new runnable and pass it to thread**

|  |
| --- |
| new Thread(      () -> System.out.println("My Runnable");  ).start(); |

**3) Sorting employees objects by their name**

|  |
| --- |
| public class LambdaIntroduction {      public static void main (String[] ar){            Employee[] employees  = {                new Employee("David"),                new Employee("Naveen"),                new Employee("Alex"),                new Employee("Richard")};              System.out.println("Before Sorting Names: "+Arrays.toString(employees));            Arrays.sort(employees, Employee::nameCompare);            System.out.println("After Sorting Names "+Arrays.toString(employees));        }  }    class Employee {    String name;      Employee(String name) {      this.name = name;    }      public static int nameCompare(Employee a1, Employee a2) {      return a1.name.compareTo(a2.name);    }      public String toString() {      return name;    }  }    Output:    Before Sorting Names: [David, Naveen, Alex, Richard]  After Sorting Names [Alex, David, Naveen, Richard] |

**4) Adding an event listener to a GUI component**

|  |
| --- |
| JButton button =  new JButton("Submit");  button.addActionListener((e) -> {      System.out.println("Click event triggered !!");  }); |

Program Examples:

**public** **class** LamdaExpressionTest {

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

// lambda expression to implement above

// functional interface. This interface

// by default implements abstractFun()

FuncInterface fobj = (**int** x) -> System.***out***.println(2 \* x);

// This calls above lambda expression and prints 10.

fobj.abstractFun(5);

}

}

// to implement a user defined functional interface.

// A sample functional interface (An interface with

// single abstract method

**interface** FuncInterface {

// An abstract function

**void** abstractFun(**int** x);

// A non-abstract (or default) function

**default** **void** normalFun() {

System.***out***.println("Hello");

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Creating an ArrayList with elements

// {1, 2, 3, 4}

ArrayList<Integer> arrL = **new** ArrayList<Integer>();

arrL.add(1);

arrL.add(2);

arrL.add(3);

arrL.add(4);

// Using lambda expression to print all elements

// of arrL

arrL.forEach(n -> System.***out***.println(n));

Note that lambda expressions can only be used to implement functional interfaces. In the above example also, the lambda expression implements [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) Functional Interface.

* The body of a lambda expression can contain zero, one or more statements.
* When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression
* When there are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned

***Stream Api In java 8:***

<https://www.baeldung.com/java-8-streams>

The Stream API is used to process collections of objects.

The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Java Stream doesn’t store data, it operates on the source data structure (collection and array) and produce pipelined data that we can use and perform specific operations. Such as we can create a stream from the list and filter it based on a condition.

Different Operations On Streams-  
**Intermediate Operations:**

1. **map:**The map method is used to map the items in the collection to other objects according to the Predicate passed as argument. (Source file definition-> Returns a stream consisting of the results of applying the given function to the elements of this stream.  
   List <Integer> number = Arrays.asList(2,3,4,5);  
   List <Integer> square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List <Integer> names = Arrays.asList("Reflection","Collection","Stream");  
   List <Integer> result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List <Integer> names = Arrays.asList("Reflection","Collection","Stream");  
   List <Integer> result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List <Integer> number = Arrays.asList(2,3,4,5,3);  
   Set <Integer> square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List <Integer> number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

Sum, min, max, average, and string concatenation are all special

cases of reduction.

List <Integer> number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

**Example:**

**public** **class** StreamAPITestClass {

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

// create a list of integers

List<Integer> number = Arrays.*asList*(2,3,4,5);

// demonstration of map method

List<Integer> square = number.stream().map(x -> x\*x).

collect(Collectors.*toList*());

System.***out***.println(square);

// create a list of String

List<String> names =

Arrays.*asList*("Reflection","Collection","Stream");

// demonstration of filter method

List<String> result = names.stream().filter(s->s.startsWith("S")).

collect(Collectors.*toList*());

System.***out***.println(result);

// demonstration of sorted method

List<String> show =

names.stream().sorted().collect(Collectors.*toList*());

List<String> showMapped =

names.stream().sorted().map(x->x.substring(0,4)).

collect(Collectors.*toList*());

System.***out***.println(show);

System.***out***.println(showMapped);

// create a list of integers

List<Integer> numbers = Arrays.*asList*(2,3,4,5,2);

// collect method returns a set

Set<Integer> squareSet =

numbers.stream().map(x->x\*x).collect(Collectors.*toSet*());

System.***out***.println(squareSet);

// demonstration of forEach method

number.stream().map(x->x\*x).forEach(y->System.***out***.println(y));

// demonstration of reduce method

**int** even =

number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

System.***out***.println(even);

}

}

* Stream is used to compute elements as per the pipelined methods without altering the original value of the object

<https://www.baeldung.com/java-8-streams> - need to go through deeply and compile the draft below.

The ***empty()*** method should be used in case of a creation of an empty stream:

|  |  |
| --- | --- |
| 1 | Stream<String> streamEmpty = Stream.empty(); |

**Stream of *Collection***

Stream can also be created of any type of *Collection*(*Collection, List, Set*):

|  |  |
| --- | --- |
| 1  2 | Collection<String> collection = Arrays.asList("a", "b", "c");  Stream<String> streamOfCollection = collection.stream(); |

Array can also be a source of a Stream:

|  |  |
| --- | --- |
| 1 | Stream<String> streamOfArray = Stream.of("a", "b", "c"); |

String[] arr = new String[]{"a", "b", "c"};

Stream<String> streamOfArrayFull = Arrays.stream(arr);

Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);

**StreamBuilder:**

Stream<String> streamBuilder =

  Stream.<String>builder().add("a").add("b").add("c").build();

**Note: See the example:**

**public** **class** StreamAPITestClass {

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

// create a list of integers

List<Integer> number = Arrays.*asList*(2, 3, 4, 5);

// demonstration of map method

List<Integer> square = number.stream().map(x -> x \* x).collect(Collectors.*toList*());

System.***out***.println(square);

// create a list of String

List<String> names = Arrays.*asList*("Reflection", "Collection", "Stream");

// demonstration of filter method

List<String> result = names.stream().filter(s -> s.startsWith("S")).collect(Collectors.*toList*());

System.***out***.println(result);

// demonstration of sorted method

List<String> show = names.stream().sorted().collect(Collectors.*toList*());

List<String> showMapped = names.stream().sorted().map(x -> x.substring(0, 4)).collect(Collectors.*toList*());

System.***out***.println(show);

System.***out***.println(showMapped);

// create a list of integers

List<Integer> numbers = Arrays.*asList*(2, 3, 4, 5, 2);

// collect method returns a set

Set<Integer> squareSet = numbers.stream().map(x -> x \* x).collect(Collectors.*toSet*());

System.***out***.println(squareSet);

// demonstration of forEach method

number.stream().map(x -> x \* x).forEach(y -> System.***out***.println("ist Line "+y));

number.stream().map(x -> x \* x).forEach(System.***out***::println);

// demonstration of reduce method

**int** even = number.stream().filter(x -> x % 2 == 0).reduce(0, (ans, i) -> ans + i);

System.***out***.println(even);

//Accumulator

Stream.*of*(10,20,22,12,14).reduce((x,y)->x+y).ifPresent(System.***out***::println);

Stream.*of*(10,20,22,12,14).reduce(Integer::*sum*).ifPresent(System.***out***::println);

Stream.*of*("java", "c", "c#", "python").reduce((x,y)->x+" | "+y).ifPresent(System.***out***::println);

//Identity & Accumulator

Integer arrSum = Stream.*of*(10,20,22,12,14).reduce(1000, (x,y)->x+y);

System.***out***.println(arrSum);

String Result = Stream.*of*("java", "c", "c#", "python").reduce("Languages:", (x,y)->x+" | "+y);

System.***out***.println(result);

//Combiner

}

}

Note: Need to study Reduce function of the stream more deeply

***Serialization In java:***

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process. **This mechanism is used to persist the object.**



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface. The ObjectOutputStream class contains **writeObject()** method for serializing an Object. The ObjectInputStream class contains **readObject()** method for deserializing an object.

**Advantages of Serialization**  
1. To save/persist state of an object.  
2. To travel an object across a network.

Serializable is a **marker interface** (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

**SerialVersionUID**  
The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization.

A Serializable class can declare its own UID explicitly by declaring a field name.  
It must be static, final and of type long.  
i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

It is also recommended to use private modifier for UID since it is not useful as inherited member.

If a serializable class doesn’t explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

**serialver**  
The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.

Points To Mark:

 -If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.

-Only non-static data members are saved via Serialization process.

-Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient

-Associated objects must be implementing Serializable interface.

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**class** Demo **implements** java.io.Serializable {

**public** **int** a;

**public** String b;

// Default constructor

**public** Demo(**int** a, String b) {

**this**.a = a;

**this**.b = b;

}

}

**public** **class** Serialzation {

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

Demo object = **new** Demo(1, "geeksforgeeks");

String filename = "file.txt";

// Serialization

**try** {

// Saving of object in a file

FileOutputStream file = **new** FileOutputStream(filename);

ObjectOutputStream out = **new** ObjectOutputStream(file);

// Method for serialization of object

out.writeObject(object);

out.close();

file.close();

System.***out***.println("Object has been serialized");

} **catch** (IOException ex) {

System.***out***.println("IOException is caught");

}

Demo object1 = **null**;

object.a= 20;

object.b= "shivam";

System.***out***.println("a = " + object.a);

System.***out***.println("b = " + object.b);

// Deserialization

**try** {

// Reading the object from a file

FileInputStream file = **new** FileInputStream(filename);

ObjectInputStream in = **new** ObjectInputStream(file);

// Method for deserialization of object

object1 = (Demo) in.readObject();

in.close();

file.close();

System.***out***.println("Object has been deserialized ");

System.***out***.println("a = " + object1.a);

System.***out***.println("b = " + object1.b);

} **catch** (IOException ex) {

System.***out***.println("IOException is caught");

} **catch** (ClassNotFoundException ex) {

System.***out***.println("ClassNotFoundException is caught");

}

}

}

//OutPut:

**Object has been serialized**

**a = 20**

**b = shivam**

**Object has been deserialized**

**a = 1**

**b = geeksforgeeks**

**Note:**

[Java Externalizable Example – More Efficient Serialization](https://howtodoinjava.com/java/serialization/java-externalizable-example/)

You can achieve more efficient serialization by implementing Externalizable interface and overriding it’s methods writeExternal() and readExternal().

***Reflection in java:*** import java.lang.reflect.\*;

Reflection in Java is a very powerful concept and it’s of little use in normal programming but it’s the backbone for most of the Java, J2EE frameworks.

**JUnit** – uses reflection to parse @Test annotation to get the test methods and then invoke it., **Spring** – [dependency injection](https://www.journaldev.com/2394/java-dependency-injection-design-pattern-example-tutorial), read more at [Spring Dependency Injection](https://www.journaldev.com/2410/spring-dependency-injection), **Tomcat** web container to forward the request to correct module by parsing their web.xml files and request URI., **Eclipse** auto completion of method names, [**Struts**](https://www.journaldev.com/struts-2), **Hibernate**

following drawbacks:

**Poor Performance**, **Security Restrictions, Security Issues, High Maintenance**

**Few code snippets:--**

Method[] methods = MyObject.class.getMethods();

for(Method method : methods){

System.out.println("method = " + method.getName());

}

Class myObjectClass = MyObject.class;

Field[] fields = myObjectClass.getFields();

Field[] fields = animalClass.getDeclaredFields();

***Annotation in java:***

***Regex in java:***

***Thread in java :***