Arrays	Collection
Arrays are Fixed in Size.	Collection are Grow able in Nature.
With Respect to Memory Arrays are Not Recommended to Use.	With Respect to Memory Collection are Recommended to Use.
With Respect to Performance Arrays are Recommended to Use.	With Respect to Performance Collection are Not Recommended to Use.
Arrays can Hold Only Homogeneous Data Elements.	Collection can Hold Both *Homogeneous* and *Heterogeneous* Elements.
Arrays can Hold Both Primitives and Objects	Collection can Hold Only Objects but Not Primitives.
Arrays Concept is Not implemented based on Some Standard Data Structure. Hence Readymade Method Support is Not Available.	For every Collection class underlying Data Structure is Available Hence Readymade Method Support is Available for Every Requirement.

ArrayList	Vector
Every Method Present Inside ArrayList is Non – Synchronized.	Every Method Present in Vector is Synchronized.
At a Time Multiple Threads are allow to Operate on ArrayList Simultaneously and Hence ArrayList Object is Not Thread Safe.	At a Time Only One Thread is allow to Operate on Vector Object and Hence Vector Object is Always Thread Safe.
Relatively Performance is High because Threads are Not required to Wait.	Relatively Performance is Low because Threads are required to Wait.
Introduced in 1.2 Version and it is Non –Legacy.	Introduced in 1.0 Version and it is Legacy.
ArrayList only supports Iterator.	Vector supports Iterator as well as Enumerator.
ArrayList increases its size 50% which is (3/2+1) of its initial capacity.	Vector increases its size 100% of its initial capacity.

Difference between Collection (I) and Collections (C):

1. Collection is an Interface which can be used to Represent a Group of Individual Objects as a Single Entity.
2. Whereas Collections is an Utility Class Present in *java.util* Package to Define Several Utility Methods for Collection Objects.

ArrayList	LinkedList
ArrayList internally uses dynamic array to store the elements.	LinkedList internally uses doubly linked list to store the elements.
Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

HashSet	LinkedHashSet
The Underlying Data Structure is Hashtable.	The Underlying Data Structure is a Combination of *LinkedList* and *Hashtable.*
Insertion Order is Not Preserved.	Insertion Order will be Preserved.
Introduced in 1.2 Version.	Introduced in 1.4 Version.

Property	Enumeration	Iterator	ListIterator
Applicable For	Only Legacy Classes	Any Collection Objects	Only List Objects
Movement	Single Direction (Only Forward)	Single Direction (Only Forward)	Bi-Direction
How To Get	By using elements()	By using iterator()	By using listIterator() of List(I)
Accessibility	Only Read	Read and Remove	Read , Remove, Replace
Methods	hasMoreElements() nextElement()	hasNext(),next(),remove()	9 Methods
Is it legacy?	Yes (1.0 Version)	No (1.2 Version)	No (1.2 Version)

Comparable	Comparator
Comparable meant for default natural sorting order.	Comparable meant for customized sorting order.
Present in java.lang package	Present in java,lang.util package
Defines Only One Method compareTo().	Contains only two Methods *compare()* and *equals()*.
All Wrapper Classes and String Class implements Comparable Interface.	The Only implemented Classes of Comparator are *Collator* and *RuleBaseCollator.*
We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method.

Property	HashSet	LinkedHashSet	TreeSet
Underlying Data Structure	Hashtable	Hashtable and LinkedList	Balanced Tree
Insertion Order	Not Preserved	Preserved	Not Preserved
Sorting Order	Not Applicable	Not Applicable	Applicable
Heterogeneous Objects	Allowed	Allowed	Not Allowed
Duplicate Objects	Not Allowed	Not Allowed	Not Allowed
null Acceptance	Allowed (Only Once)	Allowed (Only Once)	For Empty TreeSet as the 1st Element null Insertion is Possible. In all Other Cases we will get NullPointerException.

HashMap	Hashtable
No Method Present in HashMap is Synchronized.	Every Method Present in Hashtable is Synchronized.
At a Time Multiple Threads are allowed to Operate on HashMap Object simultaneously and Hence it is Not Thread Safe.	At a Time Only One Thread is allowed to Operate on the Hashtable Object and Hence it is Thread Safe.
Relatively Performance is High.	Relatively Performance is Low.
null is allowed for Both Keys and Values.	null is Not allowed for Both Keys and Values. Otherwise we will get NPE.
Introduced in 1.2 Version and it is Non – Legacy.	Introduced in 1.0 Version and it is Legacy.

HashMap	LinkedHashMap
The Underlying Data Structure is Hashtable.	The Underlying Data Structure is Combination of Hashtable and LinkedList.
Insertion is Not Preserved.	Insertion Order is Preserved.
Introduced in 1.2 Version.	Introduced in 1.4 Version.

Fail-Fast	Fail-Safe
Fail-fast iterator does not allow modification of a collection data while iterating over it.	Fail-Safe iterator allow modification of a collection data while iterating over it.
Fail-fast throws ConcurrentModificationException if a collection data is modifying while iterating over it.	Fail-safe iterator does not through ConcurrentModificationException if a collection data is modifying while iterating over it.
Original data is used for traversing the element of collection.	Copy of collection data is used for traversing the element of collection.
These iterator does not require extra memory.	These iterator does not require extra memory.
Example: ArrayList, Vector, HashMap, HashSet	Example: ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet

Relationship between .equals() method and ==(double equal operator)

1. If r1==r2 is true then r1.equals(r2) is always true i.e., if two objects are equal by== operator then these objects are always equal by .equals( ) method also.
2. If r1.equals(r2) is false then r1==r2 is always false.
3. If r1==r2 is false then we can't conclude anything about r1.equals(r2) it may return true (or) false.
4. If r1.equals(r2) is true then we can't conclude anything about r1==r2 it may returns true (or) false.

== (double equal operator)	.equals method
It is an operator applicable for both primitives and object references.	It is a method applicable only for object references but not for primitives
In the case of primitives == (double equal operator) meant for content comparison, but in the case of object references == operator meant for reference comparison.	By default .equals() method present in object class is also meant for reference comparison.
We can't override== operator for content comparison in object references.	We can override .equals() method for content comparison.
If there is no relationship between argument types then we will get compile time error saying incompatible types.(relation means child to parent or parent to child or same type)	If there is no relationship between argument types then .equals() method simply returns false and we won't get any compile time error and runtime error.
For any object reference r, r==null is always false.	For any object reference r, r.equals(null) is also returns false.

Traditional Collection	Concurrent Collection
Traditional collection is not thread safe.	Concurrent collection is always thread safe.
Even some thread safe collections are there like Vector, Hashtable etc. but performance wise not up to the mark.	In concurrent Collection performance wise performance is relatively high because of different locking mechanism used internally.
Another big problem with traditional Collection is while one thread iterating Collection, the other Thread are not allowed to modify the Collection Object simultaneously if we are trying to modify then we will get ConcurrentModificationException.	In Concurrent Collections is one thread iterating Collection, the other Thread are allowed to modify the Collection Object simultaneously in safe manner and it never throw ConcurrentModificationException. That's why it is suitable for scalable multi-Threaded Applications.

HashMap	ConcurrentHashMap
it is not thread safe.	it is Thread safe .
Relatively performance high because thread are not required to wait to operate on HashMap.	Relatively performances is low because some time Thread are required to wait to operate on ConcurrentHashMap.
While one thread iterating HashMap the other Threads are not allowed to modify Map object otherwise we will get Runtime exception saying ConcurrentModificationException.	While one Thread is iterating ConcurrentHashMap the other Thread are allowed to modify Map object in safe manner and it won't throws ConcurrentModificationException.
null is allowed for both keys and values.	null is not allowed for both keys and values .otherwise we will get NullPointerException.
introduced in 1.2 version.	introduced in 1.5 version.
Iterator of HashMap is Fail-Fast and it throws ConcurrentModificationException.	Iterator of ConcurrentHashMap is Fail-Safe and it won't throws ConcurrentModificationException.

ConcurrentHashMap	SynchronizedHashMap	Hashtable
We will get Thread safety without locking total map object just with bucket level lock.	We will get thread safety by locking whole Map Object.	We will get thread safety by locking whole Map Object.
At a time multiple thread are allowed to operate Map object in safe manner.	At a time only one Thread is allowed to perform any operation on the Map object.	At a time only one Thread is allowed to perform any operation on the Map object.
Read operation can be performed without lock but write operation can be performed with bucket level lock.	Every Read and write operation required total Map object lock.	Every Read and write operation required total Map object lock.
While one thread iterating one object ,the other thread are allowed to modify the Map and we won't get ConcurrentModificationException.	While one thread iterating one object ,the other thread are not allowed to modify the Map and otherwise we will get the ConcurrentModificationException.	While one thread iterating one object ,the other thread are not allowed to modify the Map and otherwise we will get the ConcurrentModificationException.
Iterator of ConcurrentHashMap is Fail-safe and won't throws ConcurrentModificationException.	Iterator of SynchronizedMap is Fail-Fast and it will raise the following ConcurrentModificationException.	Iterator of SynchronizedMap is Fail-Fast and it will raise the following ConcurrentModificationException.
Null is not allowed for both keys and values.	Null is allowed for both keys and values.	Null is not allowed for both keys and values.
Introduced in 1.5 version.	Introduced in 1.2 version.	Introduced in 1.0 version.

ArrayList	CopyOnWriteArrayList
It is not Thread safe .	It is not Thread safe because every update operation will be performed on separate cloned copy.
While one Thread iterating List object, the other Threads are not allowed to modify List otherwise we will get ConcurrentModificationException.	While one Thread iterating List object, the other Threads are allowed to modify List in safe manner and we will not get ConcurrentModificationException.
Iterator is Fail-Fast.	Iterator is Fail-Safe.
Iterator of ArrayList can perform remove operation.	Iterator of CopyOnWriteArrayList can not perform remove operation otherwise we will get RE:UnsupportedOperationException.
Introduced in 1.2 version.	Introduced in 1.5 version.

CopyOnWriteArrayList	syncronizedList()	vector()
We will get Thread safety because every update operation will be performed on Separate cloned copy.	We will get Thread safety because at a time List can be accessed by only one Thread at a time.	We will get Thread safety because at a time only one Thread is allowed to access Vector object.
At a time Multiple Threads are allowed to access/operate on CopyOnWriteArrayList.	At a time only one Thread is allowed to perform any operation on List object.	At a time only one Thread is allowed to perform any operation on Vector object.
While one Thread iterating List object, the other Thread are allowed to Modify and we won't get ConcurrentModification-Exception	While one thread iterating the other thread are not allowed to modify List. otherwise we will get ConcurrentModification-Exception.	While one thread iterating the other thread are not allowed to modify Vector. otherwise we will get ConcurrentModification-Exception.
Iterator is Fail-Safe and wont raise ConcurrentModification-Exception	Iterator is Fail-Fast and it will raise ConcurrentModification-Exception.	Iterator is Fail-Fast and it will raise ConcurrentModification-Exception.
Iterator cannot perform remove operation otherwise we will get UnsupportedOperationException	Iterator can perform remove operation.	Iterator can perform remove operation.
Introduced in 1.5 version	Introduced in 1.2 version	Introduced in 1.0 version

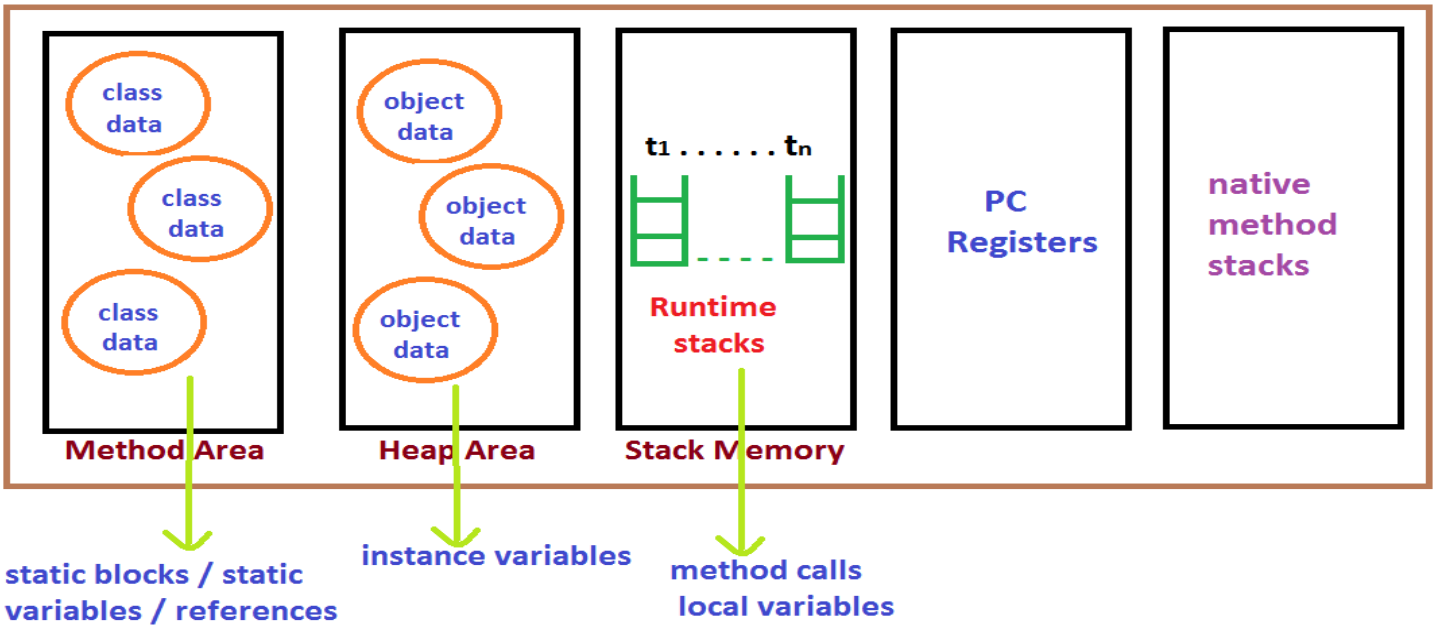
Contract between .equals() method and hashCode() method:

1. If 2 objects are equal by .equals() method compulsory their hashcodes must be equal (or) same. That is If*r1.equals(r2)* is true then *r1.hascode()==r2.hashcode( )* must be true.
2. If 2 objects are not equal by .equals() method then there are no restrictions on hashCode() methods. They may be same (or) may be different. That is If *r1.equals(r2)* is false then *r1.hashCode()==r2.hashCode()* may be same (or) may be different.
3. If hashcodes of 2 objects are equal we can't conclude anything about .equals() method it may returns true (or) false. That is If *r1.hashCode()==r2.hashCode()* is true then *r1.equals(r2)* method may returns true (or) false.
4. If hashcodes of 2 objects are not equal then these objects are always not equal by .equals() method also. That is If *r1.hashCode()==r2.hashCode()* is false then *r1.equals(r2)* is always false.

   To maintain the above contract between .equals() and hashCode() methods whenever we are overriding .equals() method compulsory we should override hashCode() method. Violation leads to no compile time error and runtime error but it is not good programming practice

String vs StringBuffer vs StringBuilder :

1. If the content is fixed and won't change frequently then we should go for String.
2. If the content will change frequently but Thread safety is required then we should go for StringBuffer.
3. If the content will change frequently and Thread safety is not required then we should go for StringBuilder.



StringBuffer	StringBuilder
Every method present in StringBuffer is synchronized.	No method present in StringBuilder is synchronized.
At a time only one thread is allow to operate on the StringBuffer object hence StringBuffer object is Thread safe.	At a time Multiple Threads are allowed to operate simultaneously on the StringBuilder object hence StringBuilder is not Thread safe.
It increases waiting time of the Thread and hence relatively performance is low.	Threads are not required to wait and hence relatively performance is high.
Introduced in 1.0 version.	Introduced in 1.5 versions.

String	StringBuffer
String class is immutable.	StringBuffer class is mutable.
String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

Serialization	Externalization
It is meant for default Serialization.	It is meant for Customized Serialization
Here every thing takes care by.	Here everything takes care by programmer and JVM.
JVM and programmer doesn't have any control	Doesn’t have any control.
Here total object will be saved always and it is not possible to save part of the object.	Here based on our requirement we can save either total object or part of the object.
Serialization is the best choice if we want to save total object to the file.	Externalization is the best choice if we want to save part of the object.
relatively performance is low	relatively performance is high
Serializable interface doesn't contain any method , and it is marker interface.	Externalizable interface contains 2 methods : 1.writeExternal() 2. readExternal() It is not a marker interface.
Serializable class not required to contains public no-arg constructor.	Externalizable class should compulsory contains public no-arg constructor otherwise we will get RuntimeException saying InvalidClassException"
transient keyword play role in serialization	transient keyword don't play any role in Externalization

& , |	&& , ||
Both arguments should be evaluated always.	Second argument evaluation is optional.
Relatively performance is low.	Relatively performance is high.
Applicable for both integral and Boolean types.	Applicable only for Boolean types but not for integral types.

new	newInstance( )
new is an operator , which can be used to create an object.	newInstance( ) is a method , present in class Class , which can be used to create an object .
We can use new operator if we know the class name at the beginning. Test t= new Test( );	We can use the newInstance( ) method , If we don't class name at the beginning and available dynamically Runtime. Object o=Class.forName(arg[0]).newInstance( );
If the corresponding .class file not available at Runtime then we will get RuntimeException saying NoClassDefFoundError , It is unchecked	If the corresponding .class file not available at Runtime then we will get RuntimeException saying ClassNotFoundException , It is checked
To used new operator the corresponding class not required to contain no argument constructor	To used newInstance( ) method the corresponding class should compulsory contain no argument constructor , Otherwise we will get RuntimeException saying InstantiationException.

Difference between ClassNotFoundException & NoClassDefFoundError :

1. For hard coded class names at Runtime in the corresponding .class files not available we will get NoClassDefFoundError , which is unchecked

   Test t = new Test( );

   In Runtime Test.class file is not available then we will get NoClassDefFoundError
2. For Dynamically provided class names at Runtime , If the corresponding .class files is not available then we will get the RuntimeException saying ClassNotFoundException

   Ex : Object o=Class.forname("Test").newInstance( );

   At Runtime if Test.class file not available then we will get the ClassNotFoundException , which is checked exception

What is the difference between instanceof and isInstance() ?

instanceof	isInstance( )
instanceof an operator which can be used to check whether the given object is particular type or not We know at the type at beginning it is available	isInstance( ) is a method , present in class Class , we can use isInstance( ) method to checked whether the given object is particular type or not We don't know at the type at beginning it is available Dynamically at Runtime.

String s = new
String("ashok");
System.out.println(s
instanceof Object);
//true
If we know the type at the beginning only.
	class Test {
	public static void main(String[] args) {
	Test t = new Test( ) ;
	System.out.println(
	Class.forName(args[0]).isInstance( ));
	//arg[0] --- We don't know the type
	at beginning
	}
	}
	java Test Test //true
	java Test String //false
	java Test Object //true

What is the difference between general import and static import ?

1. We can use normal imports to import classes and interfaces of a package. whenever we are using normal import we can access class and interfaces directly by their short name it is not require to use fully qualified names.
2. We can use static import to import static members of a particular class. whenever we are using static import it is not require to use class name we can access static members directly.

What is the difference between final and abstract ?

1. For abstract methods compulsory we should override in the child class to provide implementation. Whereas for final methods we can't override hence abstract final combination is illegal for methods.
2. For abstract classes we should compulsory create child class to provide implementation whereas for final class we can't create child class. Hence final abstract combination is illegal for classes.

What is the difference between abstract class and abstract method ?

1. If a class contain at least on abstract method then compulsory the corresponding class should be declare with abstract modifier. Because implementation is not complete and hence we can't create object of that class.
2. Even though class doesn't contain any abstract methods still we can declare the class as abstract that is an abstract class can contain zero no of abstract methods also.

Static Rules

1- Identification of static member from top to bottom.

2- Execution of static variable assignments and static block execution from top to bottom.

3- If we call any static method , inside static block and we have used some static variable but we declared it(static variable) after the static method definition that is called inside the static block then inside that method the value of the static variable will be default that is zero(0). But if this static variable initialized before static method definition then it will give the value that is assigned to it.

4- If we are trying to use any static variable direct inside the static block but we declared it after the use the it will throw an error **illegal forward reference**

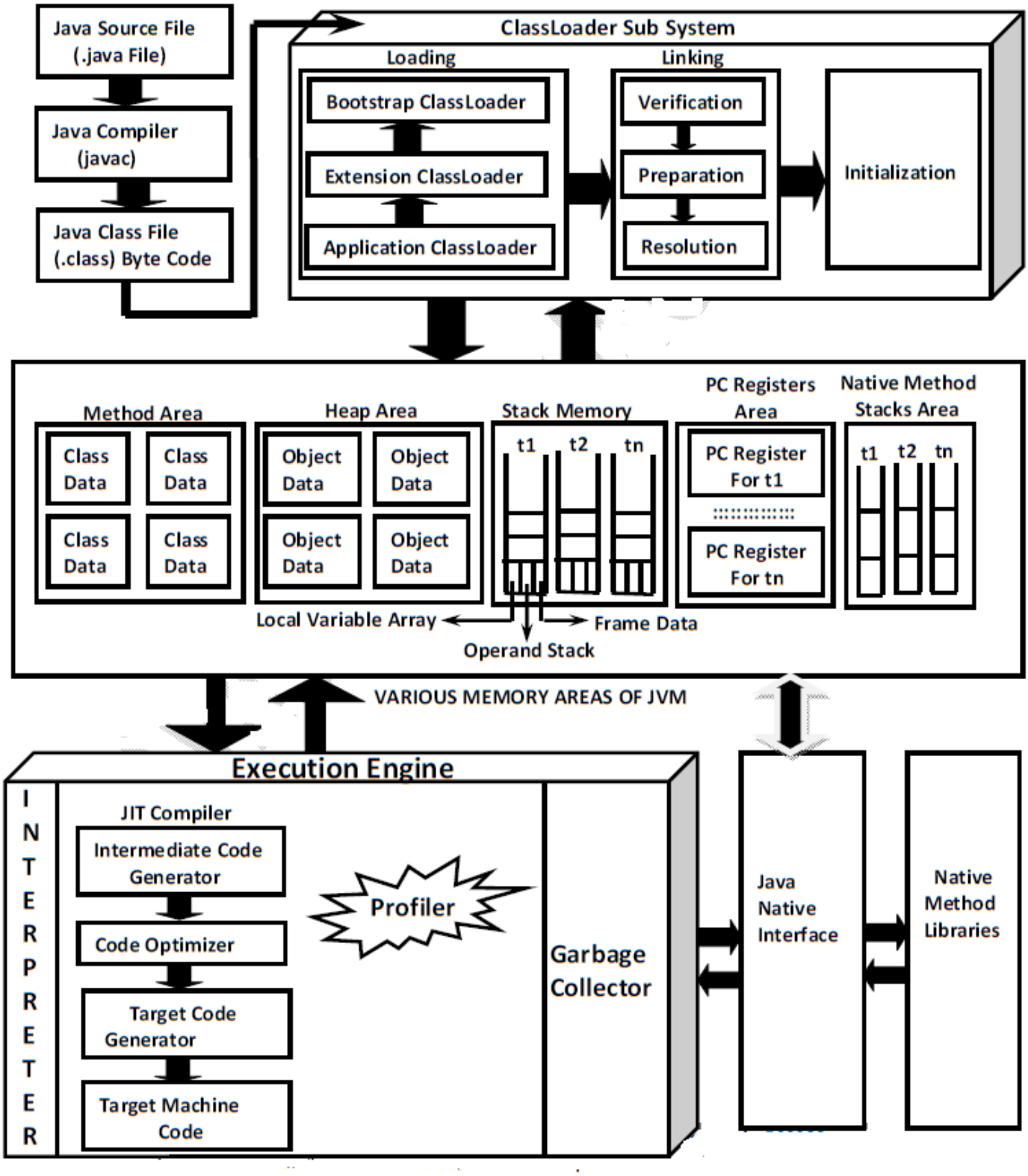
5- When all static variable and static block got executed then main method will execute.

Non-static rule

1- Non-static block execute after static member and main method.

2- If there is constructer then non-static block will execute for each time when constructer get call.

3- Non-static method execute at last.


Java Source File
(.java File)
Java Compiler
(javac)
Java Class File
(.class) Byte Code
ClassLoader Sub System
Loading
Bootstrap ClassLoader
Extension ClassLoader
Application ClassLoader
Linking
Verification
Preparation
Resolution
Initialization
Method Area
Class Data
Class Data
Class Data
Class Data
Heap Area
Object Data
Object Data
Object Data
Object Data
Stack Memory
t1
t2
tn
Local Variable Array
Frame Data
Operand Stack
PC Registers Area
PC Register For t1
PC Register For tn
Native Method Stacks Area
t1
t2
tn
VARIOUS MEMORY AREAS OF JVM
Execution Engine
INTERPRETER
JIT Compiler
Intermediate Code Generator
Code Optimizer
Target Code Generator
Target Machine Code
Profiler
Garbage Collector
Java Native Interface
Native Method Libraries

Compression of private, default, protected and public:

Visibility	private	default	protected	public
Within the same class	✓	✓	✓	✓
From child class of same package	×	✓	✓	✓
From non-child class of same package	×	✓	✓	✓
From child class of outside package	×	×	✓ (but we should use child reference only)	✓
From non-child class of outside package	×	×	×	✓

Modifier	Classes		Methods	variables	Blocks	Interfaces		enum		Constructers
	outer	Inner				Outer	Inner	outer	Inner	
public	✓	✓	✓	✓	×	✓	✓	✓	✓	✓
private	×	✓	✓	✓	×	×	✓	×	✓	✓
protected	×	✓	✓	✓	×	×	✓	×	✓	✓
<default>	✓	✓	✓	✓	×	✓	✓	✓	✓	✓
final	✓	✓	✓	✓	×	×	×	×	×	×
static	×	✓	✓	✓	✓	×	✓	×	✓	×
synchronized	×	×	✓	×	✓	×	×	×	×	×
abstract	✓	✓	✓	×	×	✓	✓	×	×	×
native	×	×	✓	×	×	×	×	×	×	×
strictfp	✓	✓	✓	×	×	✓	✓	✓	✓	×
transient	×	×	×	✓	×	×	×	×	×	×
volatile	×	×	×	✓	×	×	×	×	×	×

What is the difference between interface, abstract class and concrete class?

1. If we don't know anything about implementation just we have requirement specification then we should go for interface.
2. If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
3. If we are talking about implementation completely and ready to provide service then we should go for concrete class.

interface	Abstract class
If we don't' know anything about implementation just we have requirement specification then we should go for interface.	If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
Every method present inside interface is always public and abstract whether we are declaring or not.	Every method present inside abstract class need not be public and abstract.
We can't declare interface methods with the modifiers private, protected, final, static, synchronized, native, strictfp.	There are no restrictions on abstract class method modifiers.
Every interface variable is always public static final whether we are declaring or not.	Every abstract class variable need not be public static final.
Every interface variable is always public static final we can't declare with the following modifiers. Private, protected, transient, volatile.	There are no restrictions on abstract class variable modifiers.
For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	It is not require to perform initialization for abstract class variables at the time of declaration.
Inside interface we can't take static and instance blocks.	Inside abstract class we can take both static and instance blocks.
Inside interface we can't take constructor.	Inside abstract class we can take constructor.

Overriding	Method hiding
Both Parent and Child class methods should be non static.	Both Parent and Child class methods should be static.
Method resolution is always takes care by JVM based on runtime object.	Method resolution is always takes care by compiler based on reference type.
Overriding is also considered as runtime polymorphism (or) dynamic polymorphism (or) late binding.	Method hiding is also considered as compile time polymorphism (or) static polymorphism (or) early biding.

Property	Overloading	Overriding
Method names	Must be same.	Must be same.
Argument type	Must be different(at least order)	Must be same including order.
Method signature	Must be different.	Must be same.
Return types	No restrictions.	Must be same until 1.4v but from 1.5v onwards we can take co-variant return types also.
private, static, final methods	Can be overloaded.	Can not be overridden.
Access modifiers	No restrictions.	Weakening/reducing is not allowed.
Throws Clause	No restrictions.	If child class method throws any checked exception compulsory parent class method should throw the same checked exceptions or its parent but no restrictions for un-checked exceptions.
Method Resolution	Is always takes care by compiler based on referenced type.	Is always takes care by JVM based on runtime object.
Also known as	Compile time polymorphism (or) static(or)early binding.	Runtime polymorphism (or) dynamic (or) late binding.

super(), this()	super, this
These are constructors calls.	These are keywords.
We can use these to invoke super class & current constructors directly	We can use refers parent class and current class instance members.
We should use only inside constructors as first line, if we are using outside of constructor we will get compile time error.	We can use anywhere (i.e., instance area) except static area , other wise we will get compile time error .

Difference between final, finally, and finalize:

final:

1. final is the modifier applicable for classes, methods and variables.
2. If a class declared as the final then child class creation is not possible.
3. If a method declared as the final then overriding of that method is not possible.
4. If a variable declared as the final then reassignment is not possible

finally:

1. finally is the block always associated with try-catch to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled.

finalize:

1. finalize is a method, always invoked by Garbage Collector just before destroying an object to perform cleanup activities.

Note:

1. finally block meant for cleanup activities related to try block where as finalize() method meant for cleanup activities related to object.

2. To maintain clean up code finally block is recommended over finalize() method because we can't expect exact behavior of GC.

Thread_class constructors:

1. Thread t=new Thread();
2. Thread t=new Thread(Runnable r);
3. Thread t=new Thread(String name);
4. Thread t=new Thread(Runnable r,String name);
5. Thread t=new Thread(ThreadGroup g,String name);
6. Thread t=new Thread(ThreadGroup g,Runnable r);
7. Thread t=new Thread(ThreadGroup g,Runnable r,String name);
8. Thread t=new Thread(ThreadGroup g,Runnable r,String name,long stackSize);

Compression of yield, join and sleep () method?

Property	Yield	Join	sleep
Purpose	To pause current executing Thread for giving the chance of remaining waiting Threads of same priority.	If a Thread wants to wait until completing some other Thread then we should go for join.	If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.
Is it static?	yes	no	yes
Is it final?	no	yes	no
Is it overloaded?	No	yes	yes
Is it throws Interrupted Exception?	no	yes	yes
Is it native method?	yes	no	sleep(long ms) -->native sleep(long ms,int ns)-->non-native

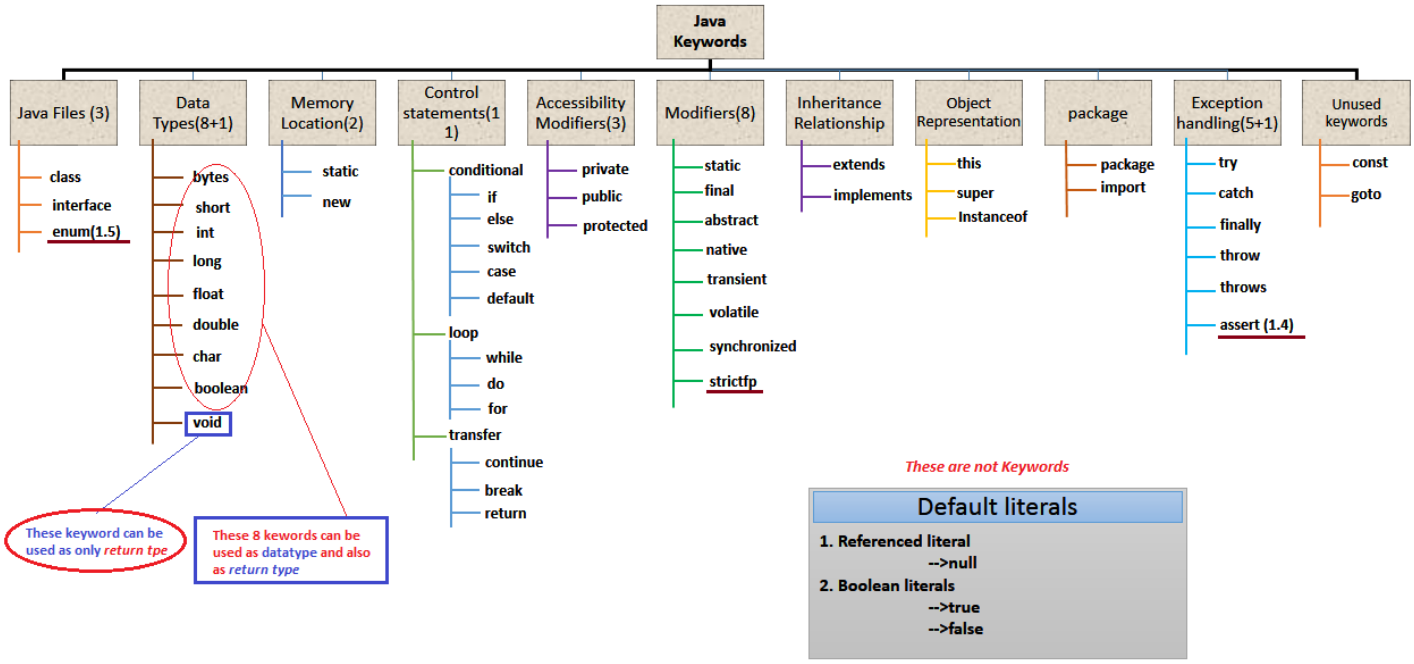
Difference between general class and anonymous inner classes:

General Class	Anonymous Inner Class
A general class can extends only one class at a time.	Of course anonymous inner class also can extends only one class at a time.
A general class can implement any no. Of interfaces at a time.	But anonymous inner class can implement only one interface at a time.
A general class can extends a class and can implement an interface simultaneously.	But anonymous inner class can extends a class or can implements an interface but not both simultaneously.
In normal Java class we can write constructor because we know name of the class.	But in anonymous inner class we can't write constructor because anonymous inner class not having any name.

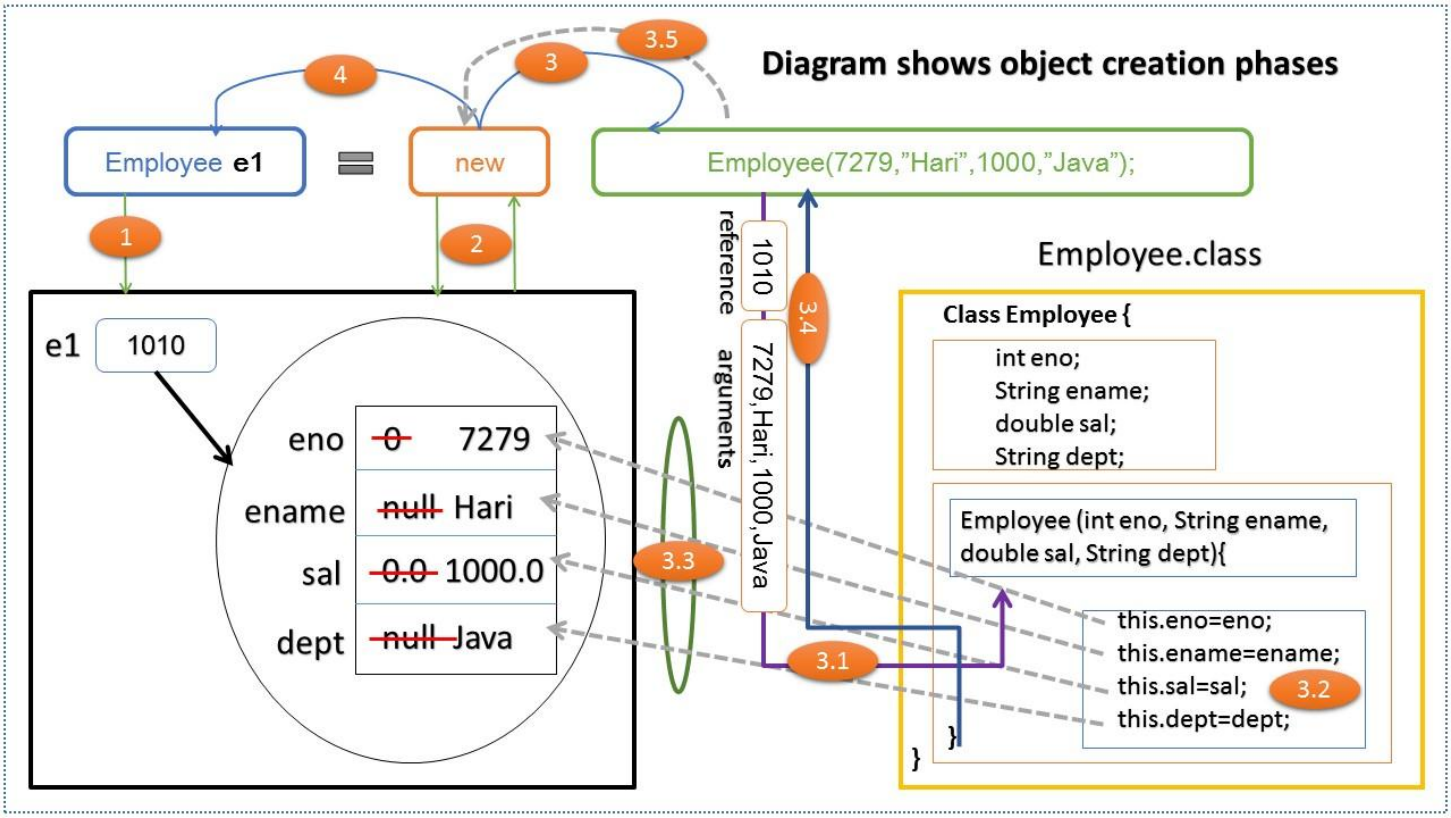
Compression between normal or regular class and static nested class ?

Normal /regular inner class	Static nested class
Without existing outer class object there is no chance of existing inner class object. That is inner class object is always associated with outer class object.	Without existing outer class object there may be a chance of existing static nested class object. That is static nested class object is not associated with outer class object.
Inside normal or regular inner class we can't declare static members.	Inside static nested class we can declare static members.
Inside normal inner class we can't declare main() method and hence we can't invoke regular inner class directly from the command prompt.	Inside static nested class we can declare main() method and hence we can invoke static nested class directly from the command prompt.
From the normal or regular inner class we can access both static and non static members of outer class directly.	From static nested class we can access only static members of outer class directly.

java Keywords Hierarchy



Sn	Method	CE	RE	Error Message
1	public static void main(String[] args)	no	no	no error
2	public static void main(String []args)	no	no	no error
3	public static void main(String args[])	no	no	no error
4	public static void main([]String args)	yes		illegal start of type
5	public static void main(String []abc)	no	no	no error
6	static public void main(String[] args)	no	no	no error
7	public static void main(String... args)	no	no	no error
8	public static int main(String[] args)	yes		missing return statement
9	public static void mian(String[] args)	no	yes	main method not found in class
10	public static void main(String[5] args)	yes		size can not be given here
11	public static void main(int[] args)	no	yes	main method not found in class
12	public static void main(String args)	no	yes	main method not found in class
13	public static void main()	no	yes	main method not found in class
14	static void main(String[] args)	no	yes	main method not found in class
15	public void main(String[] args)	no	yes	main method is not static in class
16	void main(String[] args)	no	yes	main method not found in class
17	public static final synchronized void main(String[] args)	no	no	no error
18	protected static void main(String[] args)	no	yes	main method not found in class
19	public static void main(String.* args)	yes		identifier expected
20	public static final void main(String[] args)	no	no	no error
21	private static void main(String[] args)	no	yes	main method not found in class
22	public static void main(String...[] args)	yes		identifier expected,illegal start of type
23	public static void main(String[]... args)	no	yes	main method not found in class



		Time Complexity					
		Best	Worst	Avg.	Space	Stable	Comments
Comparison Sort	Bubble Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*For each pair of indices, swap the elements if they are out of order*
	Modified Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*At each Pass check if the Array is already sorted. Best Case-Array Already sorted*
	Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	*Swap happens only when once in a Single pass*
	Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Yes	*Very small constant factor even if the complexity is O(n^2).* ***Best Case:*** *Array already sorted* ***Worst Case:*** *sorted in reverse order*
	Quick Sort	O(n.lg(n))	O(n^2)	O(n.lg(n))	O(1)	Yes	*Best Case: when pivot divide in 2 equal halves Worst Case: Array already sorted - 1/n-1 partition*
	Randomized Quick Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	Yes	*Pivot chosen randomly*
	Merge Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(n)	Yes	*Best to sort linked-list (constant extra space). Best for very large number of elements which cannot fit in memory (External sorting)*
	Heap Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	No	
Non-Comparison Sort	Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+2^k)	Yes	*k = Range of Numbers in the list*
	Radix Sort	O(n.k/s)	O(2^s.n.k/s)	O(n.k/s)	O(n)	No	
	Bucket Sort	O(n.k)	O(n^2.k)	O(n.k)	O(n.k)	Yes	