**JAVA**

1. Code editor tools useful to compile, debug and run Ex: STS, Eclipse

JVM, JRE and JDK are platform dependent but, Java is platform independent.

2. The compiled byte code is platform independent. So application developed on one platform can be run any platform like in Unix, Linux, windows.

3. JRE(Java Runtime Environment) is to run java applications. It is platform dependent (separate installation file for separate OS)

Java Applications:

1. java application: runnable file .jar

2. web applications: runnable file .war

Need to deploy runnable files in JRE to run the applications.

4. Compile Time errors can be handled by developer while developing. Editor tools show such compile time errors. (ex: syntax errors)

5. Runtime errors cannot be handled.

==================================================================================

**Java is a high level, robust, secured and object-oriented programming language.**

Where it is used:

Desktop Applications such as acrobat reader, media player, antivirus etc.

Web Applications such as irctc.co.in, javatpoint.com etc.

Enterprise Applications such as banking applications.

Mobile, Embedded System, Smart Card, Robotics, Games etc.

JVM: It is a specification that provides runtime environment in which java byte code can be executed.

It contains class loader, memory area, execution engine etc.



**1) Classloader**

Classloader is a subsystem of JVM that is used to load class files.

**2) Class(Method) Area**

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

**3) Heap**

It is the runtime data area in which objects are allocated.

**4) Stack**

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack

It contains all the native methods used in the application.

7) Execution Engine

It contains:

1) A virtual processor

2) Interpreter: Read bytecode stream then execute the instructions.

3) Just-In-Time(JIT) compiler: The JIT compiler helps improve the performance of Java programs by compiling bytecode into native machine code at run time. The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecode of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it . Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application. JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

Calling same method multiple times also requires compilation to native code but JIT directly gives native code once it got in previous step.It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

**JVM:**

(Java Virtual Machine) is an abstract machine. It is called virtual machine because it doesn't physically exist. It is a specification that provides runtime environment in which java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

The JVM performs following main tasks: Provides runtime environment

Loads code

Verifies code

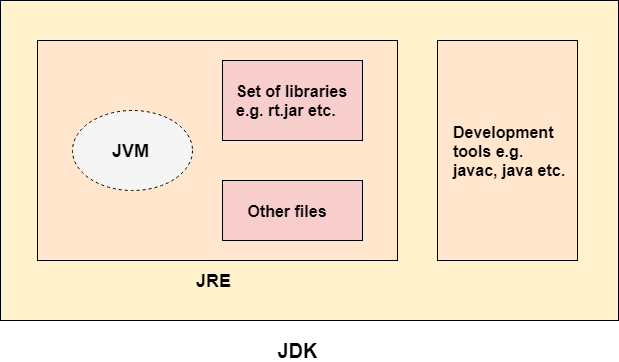
Executes code

**JRE:**

The Java Runtime Environment is a set of software tools which are used for developing java applications. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

**JDK:**

The Java Development Kit (JDK) is a software development environment which is used to develop java applications and [applets](https://www.javatpoint.com/java-applet). It physically exists. It contains JRE + development tools.



**class** A{

**static** **public** **void** main(String... args){

System.out.println("hello java4");

}

}

final strictfp public static void main(String[] args)

**Memory:**

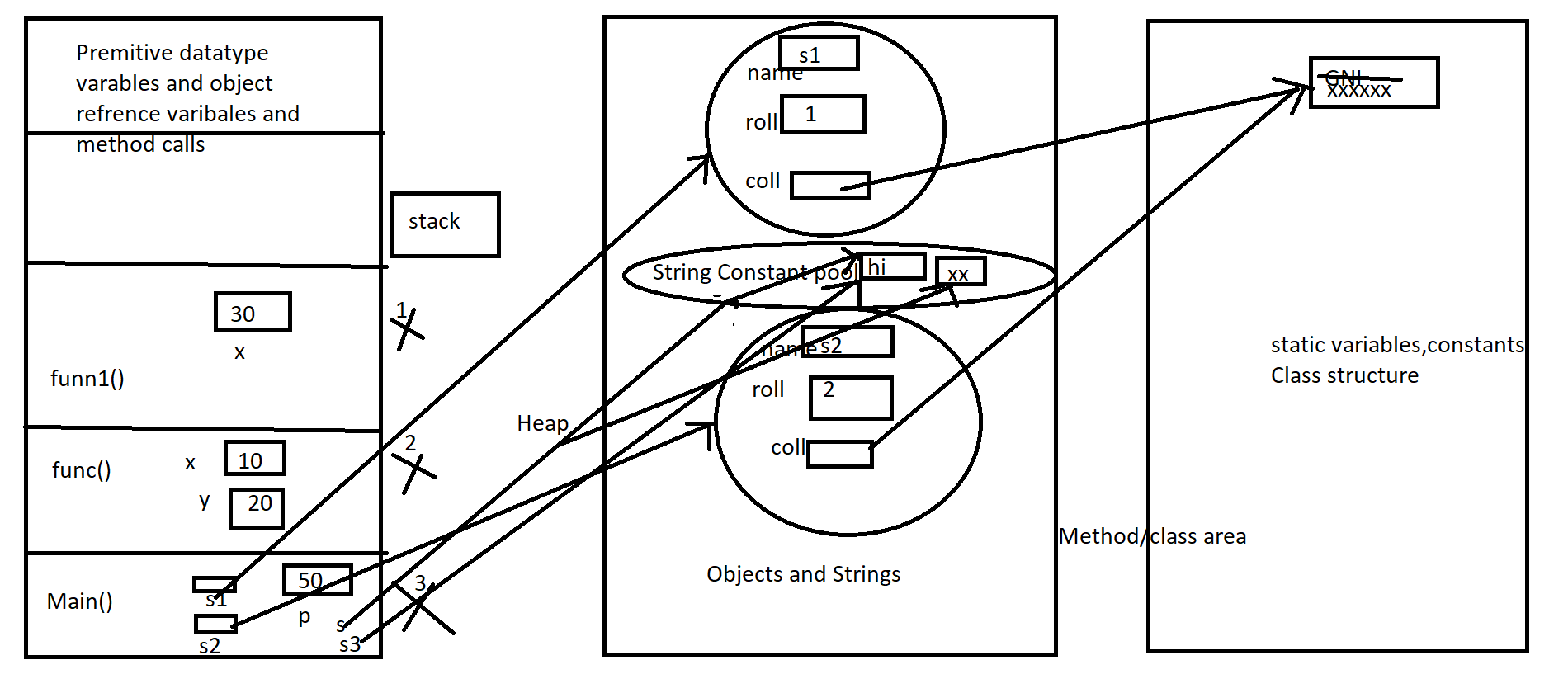
Method/class Area: Class structure, static variables, constants, runtime constant pool

Stack: contains Premitive datatypes and object refrence varibales.Method call and its method local primitive variables memory in the call stack frame (incase of Objects in method frame :only reference variables of object not whole object).

Heap: Objects and String objects (String constant pool )stored in Heap memory. and its reference in stack. Garbage collector runs on heap memory to remove unreferenced objects.

Instance (primitive) variables memory allocated in object's memory in heap.

Instance (object) variables memory allocated in heap and its reference in object's memory in heap.



When Object Created:

1.Memory allocated to hold instance varibales in Heap.

2.Objects are intialized to their default values.(by default constructor)

3.Constructor chain until Object constructor call.

4.Instace block execute before the body of the constroctor executed .

constructor()

{

super() or this();

instance block

{

}

constructor body;

}

OOPS:

1.Object

2.Class

3.Inheritance

4.Polymorphism

5.Abstraction

6.Encapsulation

**Object:** An entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Object :an instance of a class. An object contains an address and takes up some space in memory.

object's state is stored in fields and behavior is shown via methods.

Example: A dog is an object because it has states i.e. color, name, breed etc. as well as behaviors i.e. wagging the tail, barking, eating etc.

**Class:** A class can be defined as a template/blueprint that describes the behavior and state of **an object.**

Collection of objects is called as a class. It is a logical entity.

**Inheritance:** When one object acquires all the state and behaviours(methods and fields) of parent object, it is known as inheritance.

Why: It provides code reusability. It is used to achieve runtime polymorphism.

Note: The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as super class (base class, parent class).

ex: car inherits vehicle functionalities and human ->animal

**Polymorphism:** When one task performed in many ways then that is polymorphism.

Example: Draw something ==> shape or rectangle etc.

ex: a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations

ex: add integers, floats, doubles

How do we achieve in java: we use method **overloading** and method **overriding** to achieve polymorphism.

The most common use of runtime-polymorphism in OOPs : when a parent class reference is used to refer to a child class object.

Public class Deer extends Animal implements Vegetarian{

A Deer IS-A Animal

A Deer IS-A Vegetarian

A Deer IS-A Deer

A Deer IS-A Object

Deer d = new Deer(); //object creation

Animal a = d;

Vegetarian v = d;

Object o = d;

**Is-a and Has-a:** Car is a vehicle and it has an Engine

**Abstraction:**Hiding internal implimentation details from the user and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

How do we achieve in java: we use abstract classes and interfaces to achieve abstraction.

We can achieve 100% abstraction using interfaces.

1) If we don’t know about implementation just we have to represent the specification then we should

go for interface

2) If we don’t know about complete implementation just we have partial implementation then we

should go for abstract.

3) If we know complete implementation and if we r ready to provide service then we should go for

concrete class.

ex:ATM machine for cash withdrawal

**Encapsulation:** Binding the data(variables) and functions together into a single unit inorder to protect from un-authorized access is known as encapsulation.

How do we achieve in java: Declare the variables of a class as private and provide public setter and getter methods to modify and view the variables values. Ex:Java Bean.

ex: If your balance variable is declared as a public variable in the bank software,The person who has to see his account balance, will have to access only private members through methods defined inside that class and this method will ask your account holder name or user Id, and password for authentication.

ex: cola vending machine

In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

**Method Overloading:** Definingfunctions with the same name and with different number of arguments or different type of arguments.

**Method overloading increases the readability of the program and compile time polymerphism.**

There are two ways to overload the method in java

1.By changing number of arguments

2.By changing the data type

Can we overload java main() method? :YES

Note: In java, Method Overloading is not possible by changing the return type of the method only.

1.By changing number of arguments **:**

class TestOverloading1{

public static void main(String[] args){

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(11,11,11));

}

static int add(int a,int b){

return a+b;

}

static int add(int a,int b,int c){

return a+b+c;

}

}

2.By changing the data type:

class TestOverloading2{

public static void main(String[] args){

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(12.3,12.6));

}

static int add(int a, int b){

return a+b;

}

static double add(double a, double b){

return a+b;

}

}

Can we overload java main() method?

**Yes**, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

class TestOverloading4{

public static void main(String[] args){

**System.out.println("main with String[]");**

}

public static void main(String args){

System.out.println("main with String");

}

public static void main(){

System.out.println("main without args");

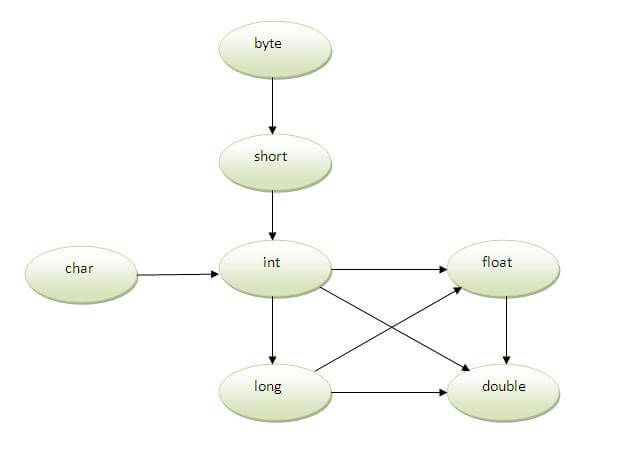
}

}

Output: main with String**[]**

**Method Overloading and Type Promotion:**

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

void sum(int a,long b){System.out.println(a+b);}

obj.sum(20,20)

Example of Method Overloading with Type Promotion if matching found:

void sum(int a,int b){System.out.println(a+b);}

void sum(long a,long b){System.out.println(a+b);}

obj.sum(20,20)

### Example of Method Overloading with Type Promotion in case of ambiguity:

1. void sum(int a,long b){System.out.println("a method invoked");}
2. void sum(long a,int b){System.out.println("b method invoked");}
3. obj.sum(20,20);//now ambiguity

**Method Overriding: Defining same function in subclass which is already defined or declared in super class. Should have same function name and same number of arguments and same type of arguments and same return type.**

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

### Usage of Java Method Overriding

### Method overriding is used to provide specific implementation of a method that is already provided by its super class.

### Method overriding is used for runtime polymorphism

### Rules for Java Method Overriding

### method must have same name as in the parent class

### method must have same parameter as in the parent class.

### must be IS-A relationship (inheritance).

### Problem without method overriding: I have to provide a specific implementation of run() method in subclass.

### Example of method overriding:

1. **class** Vehicle{
2. **void** run(){System.out.println("Vehicle is running");}
3. }
4. **class** Bike2 **extends** Vehicle{
5. **void** run(){System.out.println("Bike is running safely");}
6. **public** **static** **void** main(String args[]){
7. Bike2 obj = **new** Bike2();
8. obj.run();
9. }

### Java method overriding example of bank

### Can we override static method? No, static method cannot be overridden. It can be proved by runtime polymorphism

Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

Can we override java main method? No, because main is a static method.

# Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method Overloading** | **Method Overriding** |
| 1) | Method overloading is used to increase the readability of the program. | Method overriding is used to provide the specific implementation of the method that is already provided by its super class. |
| 2) | Method overloading is performed within class. | Method overriding occurs in two classes that have IS-A (inheritance) relationship. |
| 3) | In case of method overloading, parameter must be different. | In case of method overriding, parameter must be same. |
| 4) | Method overloading is the example of compile time polymorphism. | Method overriding is the example of run time polymorphism. |
| 5) | In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter. | Return type must be same or covariant in method overriding. |

### compile time polymorphism: the method that need to be called is decided at compile time.

### run time polymorphism: the method that need to be called is decided at run time.

**Exception:**

java.lang.Throwable

EXCEPTIONHANDLING:handling of runtime errors to maintain the normal flow execution.

Exception:is an event that distrubs the noral flow of execution.It is an object which is thrown at runtime.

Exceptions:IOEx,SQlEx,ClassNotfoundEx,RuntimeEx==>ArithmaticEx,NullPointerEx,NumberFormatEx,IndexOutOfBoundsEx==>String,Array

Errors:StackOverflowError,OoutofMemoryError,VirtualMachineError,Class def not found

Types of Java Exceptions:Checked,Un-checked,error(un-checked).

Whether the exception is checked or unchecked it always occur at runtime only.

**Checked Exception(Compile Time Exceptions):classes which directly inherit Throwable class except RuntimeEx and Error are known as Checked Exeption**

**checked by the compiler for smooth execution** of the program at runtime

Ex:IOException, SQLException etc.Checked exceptions are checked at compile-time.

Need to resolve at compile time ,utill that comipler will not allow us to compile.

Unchecked Exception:classes which inherit RuntimeException are known as unchecked exceptions

Ex:ArithmeticException, NullPointerExcepti-on, ArrayIndexOutOfBoundsException etc.Unchecked exceptions are not checked at compile-time,but they are checked at runtime.

Error:Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Keywords:

try:Enclose the code that might throw an exception.try must be follwed by either catch or finally.can't use try block alone.

catch:To handle the exception,preceded by try block, we can't use catch block alone.It can be followed by finally block later

-->You can use multiple catch block with a single try.

finally:To execute the important code(cleanup)of the program.It is executed whether an exception is handled or not.

-->Ex:closing connection,stream.If any Ex with no valid catch then Finally{}==>ExceptionDesc==>stacktrace==>terminate

Internal working:Exception Object thrown==>Prints exceptionDesc==>prints stacktrace==>terminate the program

Rule: At a time only one Exception is occured and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general EX:catch for ArithmeticException must come before catch for Exception

Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

throw:To throw an exception excplicitly.

-->can throw either checked or uncheked exception.mainly used to throw custom exception.

-->throw new IOException("sorry device error); throw new CustomException("custom ex description")

Java Exception propagation:call stack order

Rule:By default Unchecked Exceptions are forwarded in calling chain(propagated).

Rule:By default,Checked Exceptions are not forwarded in calling chain(propagated).use throws in declaration otherwise compileTime error.

throws:To declare exceptions.It specifies that there may occur an exception in the method.It is always used with method signature.

return\_type method\_name() throws exception\_class\_name.You can declare multiple exceptions

Rule: If you are calling a method that declares an exception,you must either caught or declare the exception.

Final is used to apply restrictions on class, method and variable.Final class can't be inherited, final method can't be overridden and final variable value can't be changed.

Finally is used to place important code, it will be executed whether exception is handled or not.

Finalize is used to perform clean up processing just before object is garbage collected.

Final is a keyword.

Finally is a block.

Finalize is a method.

ExceptionHandling with MethodOverriding:

If the superclass method does not declare an exception:

-->If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

If the superclass method declares an exception:

-->If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the parent class constructor throws some checked exception then the child class constructor must throw same checked exception or its parent other wise compiler error.

Custom:Exception:

class InvalidAgeException extends Exception{

InvalidAgeException(String s){

super(s);

}

}

Exception due to program code but Error due to lack of system resources.

ClassNotFoundException is an exception that occurs when you try to load a class at run time using Class.forName() or loadClass() methods and mentioned classes are not found in the classpath.

NoClassDefFoundError is an error that occurs when a particular class is present at compile time, but was missing at run time.

Throwable class contains the following methods to display error information:

printStackTrace: It displays error information in the following format.<==>Name of Exception Class : Description,StackTace

toString: it displays error in the following format.<==>Name of Exception Class : Description

getMessage:it displays error information in the following format<==>Description

If there is no chance of raising an exception in try statement then we r not allowed to maintain catch block for that exception violation leads to compile time error but this rule is applicable only for fully checked exceptions.Ex:IOException

statements after finally will be executed if exception handled with proper exception Handler.

OFF

FATAL

ERROR

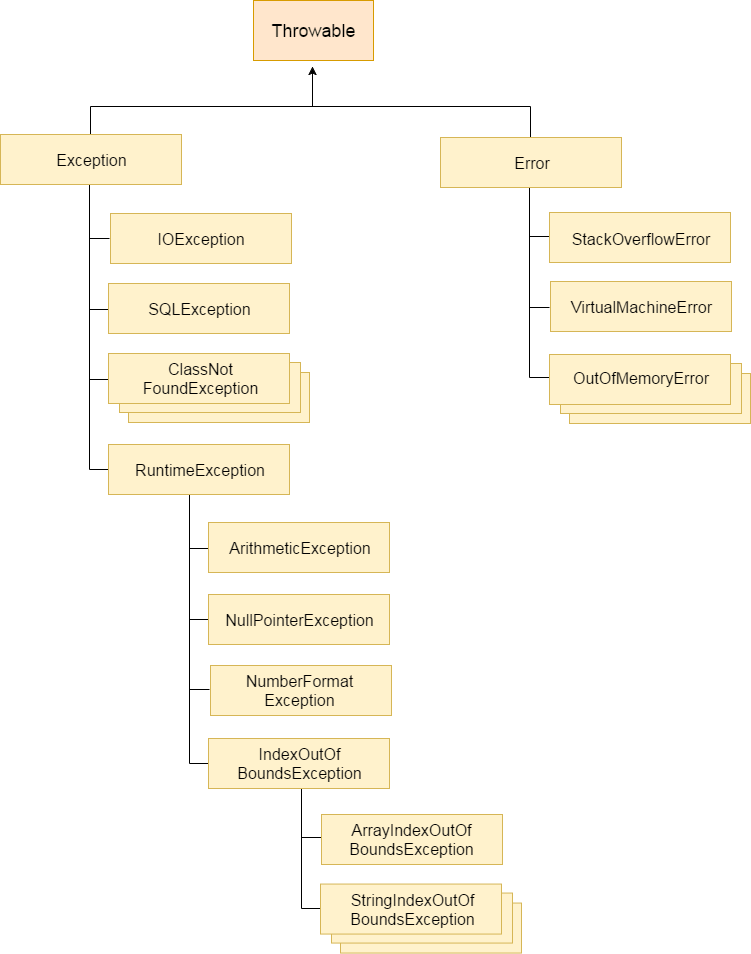
WARN

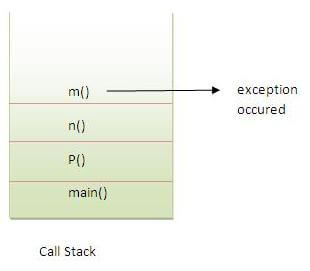
INFO

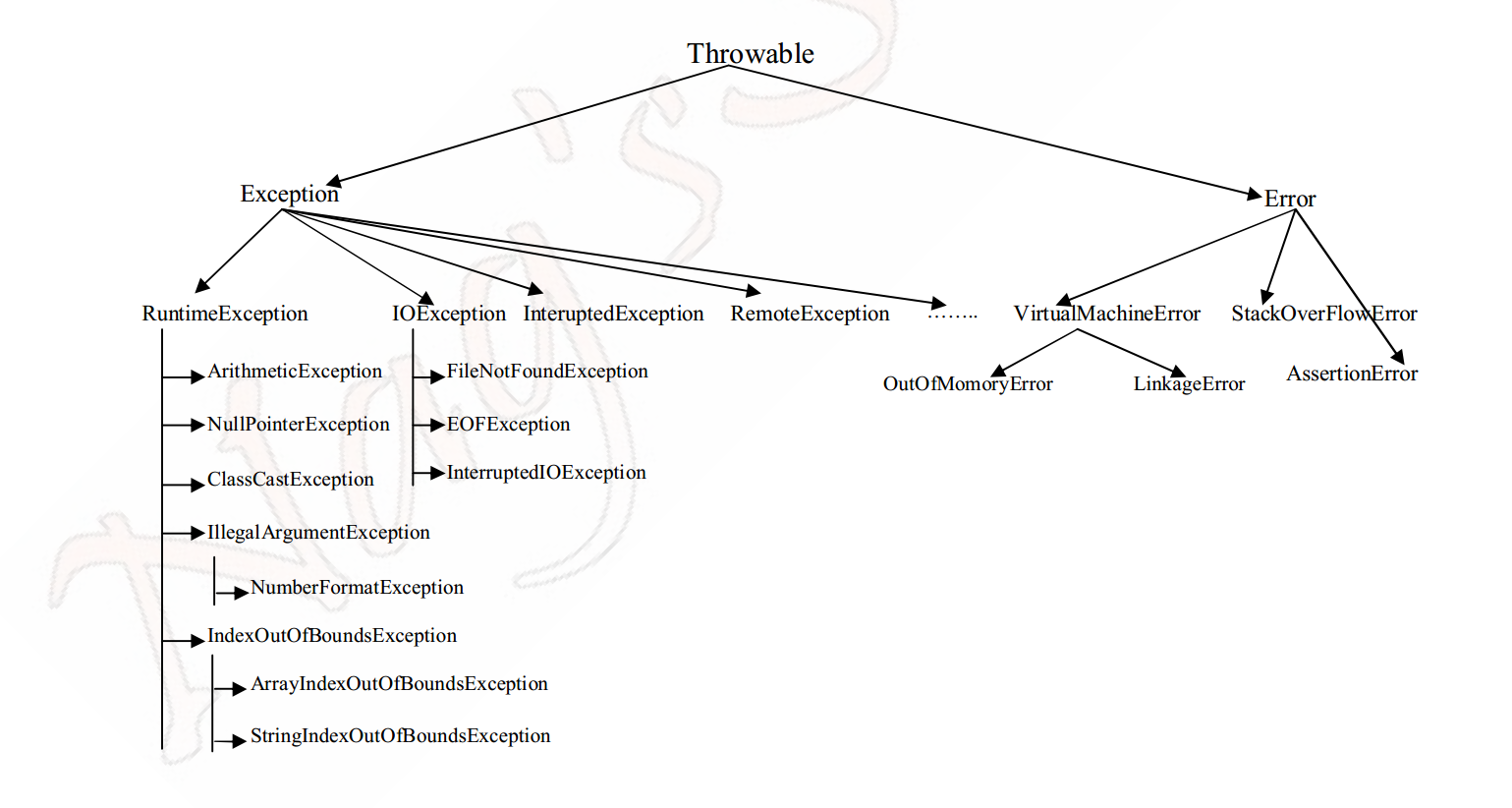
DEBUG

TRACE

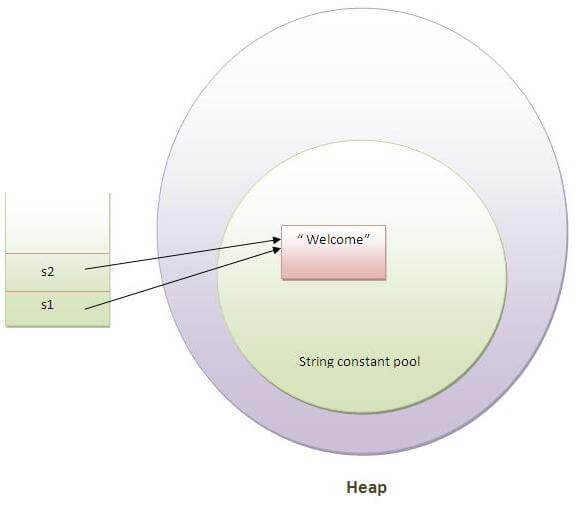
ALL



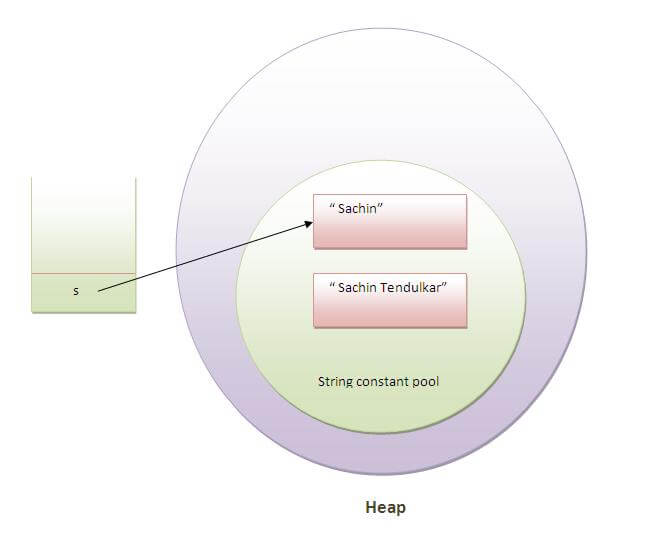




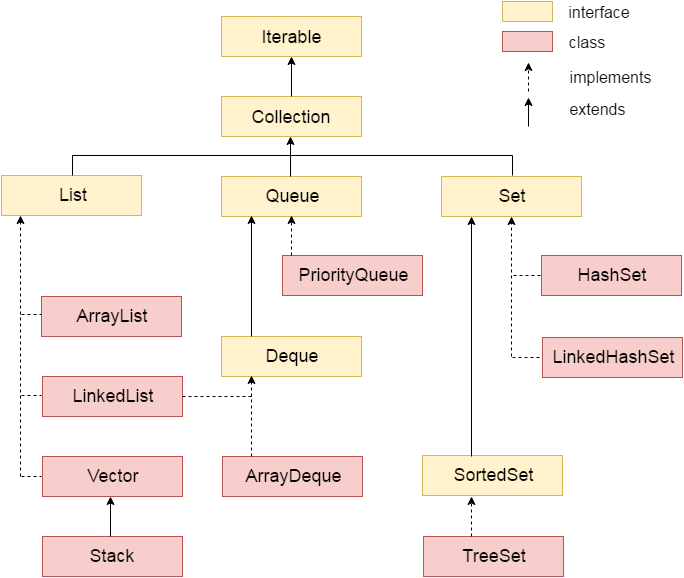
1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance

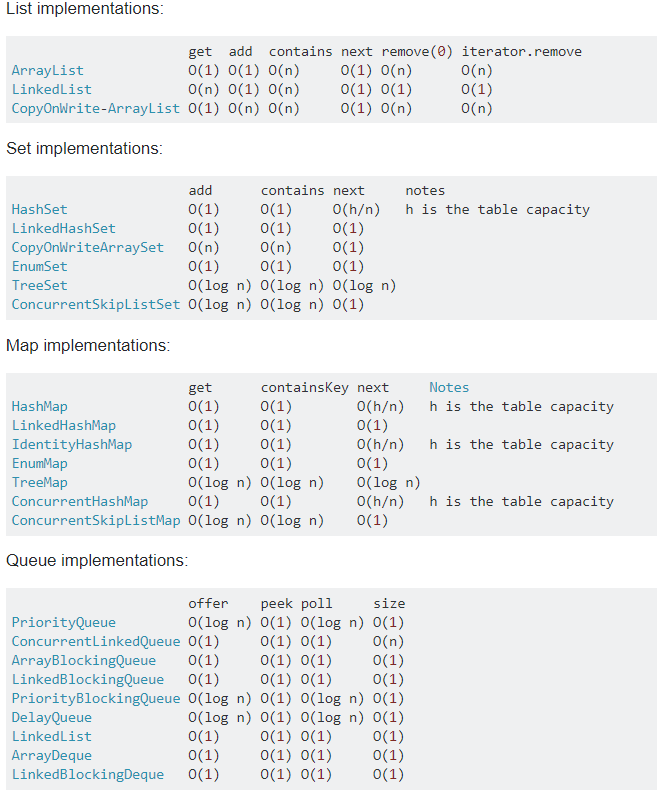


1. String s="Sachin";
2. s.concat(" Tendulkar")



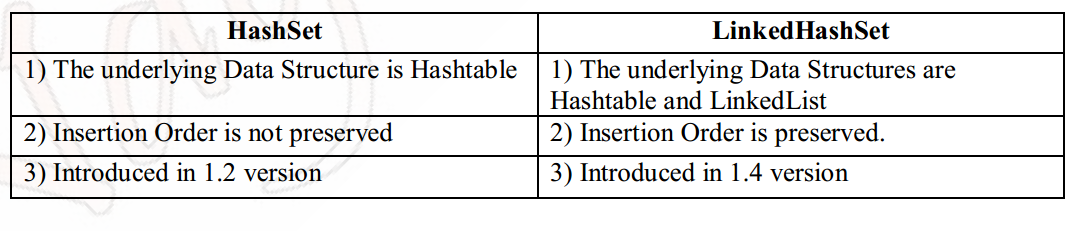
Collections:





ArrayList vs LinkedList:

|  |  |  |
| --- | --- | --- |
|  | **ArrayList** | **LinkedList** |
| Description | Dynamic array | Doubly-linked list |
| Random Access with index (get()) | O(1) | O(n) |
| Insertion (add()) / removal at the back | amortized O(1) | O(1) |
| Insertion / removal at the front | O(n) | O(1) |
| One step of iteration through an Iterator | O(1) | O(1) |
| Insertion / removal in the middle through an Iterator / ListIterator | O(n) | O(1) |
| Insertion / removal in the middle through the index | O(n) | O(n) |
| Search contains() / removal remove() by object | O(n) | O(n) |



|  |  |  |
| --- | --- | --- |
| **HashMap** | **TreeMap** | **LinkedHashMap** |
| 1) HashMap can contain one null key. | TreeMap can not contain any null key. | LinkedHashMap can contain one null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. | LinkedHashMap maintains insertion order. |

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class.Fhas |
| **ArrayList** | **Vector** |
| 1) ArrayList is not synchronized. | Vector is synchronized. |
| 2) ArrayList increments 50% of current array size if number of element exceeds from its capacity. | Vector increments 100% means doubles the array size if total number of element exceeds than its capacity. |
| 3) ArrayList is not a legacy class, it is introduced in JDK 1.2. | Vector is a legacy class. |
| 4) ArrayList is fast because it is non-synchronized. | Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses Iterator interface to traverse the elements. | Vector uses Enumeration interface to traverse the elements. But it can use Iterator also. |

In Wrapper classes and String class both equals and hashCode methods overrided.So content comparision will be performed in add and remove.(not object comparision)

Collections in Java:(java.util package)

Is a framework that provides an architecture to store and manipulate the group of objects(single unit of objects i.e group)

operations that can be performed on data are:searching, sorting, insertion, manipulation, deletion.

methods:equals(),hashCode(),add(All),remove(All),clear(),isempty(),size(),iterator(),contains(All),retain(All)

Iterator interface(super interface):to iterate forward direction only.hasNext(),next(),remove();

**ArrayList(size:10)**:uses a dynamic array for storing the elements.implements List interface.

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

Java ArrayList class can contain **duplicate** elements.

Java ArrayList class maintains **insertion order**.

**'null’** insertion is possible.

capacity = **(current capacity \* 3/2) + 1**

Java ArrayList class is **non synchronized**.(not thread-safe)

Java ArrayList allows random access because array works at the **index basis**.

ArrayList is **best** suitable if our frequent operation is **retrieval** operation

Adv:ArrayList is better for **storing and accessing data**.

Drawback:(insert or delete an element in the middle:O(n))manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

**Best**: **retrieve**(bcz index based) , add last, remove last. **Worst**: add first, remove first.

O(1) get()

O(1) add()-->O(n) incase of size full and need to create big array and copy all the values and append new one.

O(n) add(index,object)🡪 2\*O(n)

O(n) remove()

**Boolean add**(E e), void add(int index, E element), contains(Object o), **get**(int index), indexOf(Object o), E **remove**(int index), boolean remove(Object o)

Generic collection:allows you to have only one type of object in collection.It is type safe so typecasting is not required at run time.

**LinkedList**:uses doubly linked list to store the elements.

**Best**: add first, last & remove first, last. **Worst**: Retrieve, need to traverse all the nodes.

Java LinkedList class can contain duplicate elements.

Java LinkedList class maintains insertion order.

‘null’ insertion is possible.

Java LinkedList class is non synchronized.

**Adv**:In Java LinkedList class, **manipulation is fast because no shifting needs** to be occurred.

**bestchoice** if our frequent operation is **insertion or deletion in the middle**(no shift operations are required)

LinkedList is the **worst** choice if our frequent operation is **retrieval** operation

Java LinkedList class can be used as list, **stack** or **queue**.

O(n) get()

O(1) add()

O(1) remove()

**add**(),addFirst,addLast,**get**(),getFirst,getLast,**remove**(),removeFirst,reoveLast,peekFirst,peekLast,pollFirst,PollLast,peek,push,pop

VectorClass:(10)

The underlying Data structure for the vector is resizable array or growable array.

Insertion order is preserved.

Duplicate objects are allowed.

‘null’ insertion is possible.

Heterogeneous objects are allowed.

new capacity = 2 \* current capacity

Best choice if the frequent operation is retrieval.

Worst choice if the frequent operation is insertion or deletion in the middle.

Vector class implemented serializable, cloneable and RandomAccess Interfaces.

Vector is synchronized

Cursors in Collection:

1. Enumeration:hasMoreElements(),nextElement()

2. Iterator:hasNext(),next(),remove();

3. ListIterator:To traverse the element in backward and forward direction.hasNext(),next(),hasPrevious(),previous();

Set:duplicate objects are not allowed.

HashSet(16):uses a hash table(**hashmap**) for storage.

HashSet stores the elements by using a mechanism called hashing.

HashSet contains unique elements only.

does not mainatain insertion order.

‘null’ insertion is possible

capacity = 0.75 \* current capacity

HashSet is the best choice if the frequent operation is **Search** Operation

Add(Object),remove(object),contains(object).

1. HashSet doesn't allow duplicate elements.
2. It allows at most one null element.
3. **It internally uses HashMap as backing data structure and stores private static final Object class instance PRESENT as a value of all the keys. As we are storing same static field as value to all the keys, we can neglect additional space being used by this value field.**
4. **HashSet's add() method uses HashMap's**[**put()**](https://codepumpkin.com/?p=855/#put)**method internally.**
5. **As it is backed by HashMap, we must override**[**hashCode()**](https://codepumpkin.com/?p=855/#HashCode)**and**[**equals()**](https://codepumpkin.com/?p=855/#Equals)**method while creating HashSet of custom class objects. Refer our article**[**How does HashMap work internally in Java?**](https://codepumpkin.com/?p=855)**to know more about HashMap internal implementation.**
6. **HashSet is not thread-safe. Here are some of the thread-safe implementation of Set:**
7. –CopyOnWriteArraySet  
   –Collections.synchronizedSet(Set set)  
   –ConcurrentSkipListSet  
   –​Collections.newSetFromMap(new ConcurrentHashMap())
8. HashSet doesn't maintain insertion order. To maintain insertion or access order, we need to use LinkedHashSet.
9. HashSet also doesn't store elements in sorted order. Instead we should use TreeSet to store elements in sorted order.
10. Similar to HashSet, LinkedHashSet has also been implemented using LinkedHashMap and TreeSet is implemented using TreeMap. And all these data structure uses private static final Object class instance PRESENT as a value of all their keys.

Difference between List and Set:

List can contain duplicate elements whereas Set contains unique elements only.

**LinkedHashSet:**Hash table and Linked list implementation of the set interface.(**linkedHashMap**)

Contains unique elements only like HashSet.

Provides all optional set operations, and permits null elements.

**Maintains insertion order.**

For implementing **caching application** the best suitable Data structure is **LinkedHashSet and**

**LinkedHashMap** where duplicate objects are not allowed and insertion order Must be preserved.

**TreeSet**:uses binary search trees(red-black trees)(**Balanced Tree**)

Contains unique elements only like HashSet.

Adv:Access and retrieval times are quiet fast.

Maintains ascending order.

For the empty TreeSet as the first element null insertion is possible. But after inserting null if we are

trying to insert any other element we will get **NullPointerException**.

System.out.println**("A".compareTo(null)**);  **NullPointerException**

If the TreeSet already contains some elements if we are trying to insert null we will get NullPointerException

If we are depending on natural sorting order compulsory the objects should be **homogeneous** and

**comparable** other wise we will get **class** **cast Exception-runtime**.

An object is said to be comparable(if and only if) the corresponding class has to implement

comparable interface.

Note: while Inserting the objects into the TreeSet JVM internally uses compareTo() method if we are depending on natural sorting order.

Sometimes we have to define our own customized sorting order, then we should go for comparator Interface

TreeSet t = new TreeSet();

t.add(new StringBuffer("A"));

All wrapper classes and String class already implemented comparable interface. But the String buffer

doesn’t implement comparable interface. Hence in the above program we got class cast exception.

**CopyOnWriteArraySet**:

* The internal implementation of CopyOnWriteArraySet is CopyOnWriteArrayList only.
* Insertion order is preserved and duplicates are not-allowed.
* Multiple Threads are able to perform update operation simultaneously but for every update operation a **separate cloned copy** is created. As for every update a new cloned copy will be created which is costly. Hence if multiple update operation are required then it is not recommended to use CopyOnWriteArraySet.
* While one thread iterating the Set, other threads can perform updation, here we wont get any runtime exception like ConcurrentModificationException.
* Iterator of CopyOnWriteArraySet class can perform only read only and wont perform deletion, otherwise we will get Run-time exception UnsupportedOperationException.

**Map**:

Each key and value pair is known as an Entry. Map contains only unique keys. Map is useful if you have to search,update or delete elements on the basis of key.

Map doesn't allow duplicate keys, but you can have duplicate values.

HashMap and LinkedHashMap allows null keys and values but TreeMap doesn't allow any null key or value.

Map.Entry<K,V> Interface:getKey(),getValue()

Set<Map.Entry<K,V>> m=map.entrySet();.keySet();.values();

Class Node<K,V>{

int hashCode;

K k;

V v;

Node<K,V> next;

//for linkedHashMap before and after present to track order

}

V put(K k,V v){

int hashCode=hashCode(k){ var0 == **null** ? 0 : (var1 = var0.hashCode()) ^ var1 >>> 16;}

int index=hashCode & n-1;

HashTable.put(index,Node<K,V>);or Entry[i]=new Entry(key,value,hashcode,next)

}

V get(K k){

int hashCode=hashCode(k)

int index=hashCode & n-1;

return HashTable(index);

}

HashMap:uses HashTable

It doesn't maintain any order.

A HashMap contains values based on the key.

It contains only unique elements.

It may have one null key and multiple null values.

It maintains no order.

If you will be adding and removing elements a lot,it will be better to use HashMap, because LinkedHashMap will be slower to do those operations(need extra storage to maintain previous and next node addresses).

LinkedHashMap:uses Hash table and Linked list implementation of the Map interface

**It maintains insertion order.**

A LinkedHashMap contains values based on the key.

It contains only unique elements.

It may have one null key and multiple null values.

It is same as HashMap instead maintains insertion order.

is useful for implementing a LRU (least recently used) caching policy.

if you will be **iterating heavily**, it may be a good idea to use this.(iterate faster)

TreeMap: uses binary search trees(red-black trees)

it maintains **ascending** order of keys.

It contains only unique elements.

It cannot have **null key** but can have multiple null values.

It is same as HashMap instead maintains ascending order and dont have null key.

**Can I paSS a custom Object as key to Treemap ? by default not ,we need to make the Objects comparable by using comparator or comparable.**

Difference b/w comparable AND comparator: **Single sorting sequence and multi sorting sequence.**

Hashtable:

A Hashtable is an array of list. Each list is known as a bucket(node).The position of bucket is identified by calling the hashcode() method. A Hashtable contains val,2ues based on the key.

It contains only unique elements.

It may have not have any null key or value.

It is synchronized.

**Key points of ConcurrentHashMap:**

* The underlined data structure for ConcurrentHashMap is Hashtable.
* ConcurrentHashMap class is thread-safe i.e. multiple thread can operate on a single object without any complications.
* At a time any number of threads are applicable for read operation without locking the ConcurrentHashMap object which is not there in HashMap.
* In ConcurrentHashMap, the Object is divided into number of segments according to the concurrency level.
* Default concurrency-level of ConcurrentHashMap is 16.
* In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updation in object, thread must lock the particular segment in which thread want to operate.This type of locking mechanism is known as **Segment locking or bucket locking**.Hence at a time 16 updation operations can be performed by threads.
* null insertion is not possible in ConcurrentHashMap as key or value.

Iterators in java are used to iterate over the Collection objects.

**Fail-Fas**t: if a thread iterating over a collection and other thread modify the collection at the same time then if it throws ConcurrentModificationException then it is called fail fast.

iterators immediately throw ConcurrentModificationException if there is structural modification of the collection.

Structural modification means adding, removing or updating any element from collection while a thread is iterating over that collection.

**Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.**

Fail-Safe iterators don’t throw any exceptions if **a collection is structurally modified while iterating over it.**

This is because, they operate on the clone of the collection, not on the original collection and that’s why they are called fail-safe iterators.

Any structural modification done to the iterator affects the copied collection,not original collection.So,original collection remains structurally unchanged.

**Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.**

Collections:It is a util class conatins static methods like sort.

**Sorting** in Collection:sort(List list):

We can sort the elements of:

String objects

Wrapper class objects

User-defined class objects

If collection elements are of Set or Map, we can use **TreeSet or TreeMap**.,But **We cannot sort the elements of List.**

Collections class provides methods for sorting the elements of List type elements.

public void sort(List list): is used to sort the elements of List.**List elements must be of Comparable type.**

Note: **String class and Wrapper classes implements the Comparable interface**.So if you store the objects of string or wrapper classes, it will be Comparable.

**Comparable** interface: is used to order the objects of user-defined class. contains only one method named **compareTo**(Object).

provides **single sorting sequence** only.i.e.you can sort the elements on based on single data member only.Ex:may be rollno, name, age or anything else.

class **Student** implements **Comparable**<Student>{

int age;

public int compareTo(Student st){

if(this.age==st.age)

return 0;

else if(age>st.age)

return 1;

else

return -1;

}}

Comparator interface: sort(List list, Comparator c):

2 methods compare(Object obj1,Object obj2)==must implement and equals(Object element)==optional.

It provides **multiple sorting sequence** i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

return I1.compareTo(I2);  [0,5,10,15,20]

return I2.compareTo(I1);  [20,15,10,5,0]

return -I1.compareTo(I2);  [20,15,10,5,0]

return -I2.compareTo(I1);  [0,5,10,15,20]

return I2-I1;  [20,15,10,5,0]

return I1-I2;  [0,5,10,15,20]

public int compare(Object obj1, Object obj2)

{

return -1;  [0,20,15,5,10] (reverse of Insertion order)

return 1;  [10,5,15,20,0] (Insertion Order)

return 0;  [10] (All the remaining elements considered as duplicate objects)

}

**GARBAGE COLLECTION:**

The ways to make an object **eligible for Garbage Collector:**

**1.Nullifying the reference Variable**

**2.Reassigning the reference Variable**

**3.The Objects Created inside a method**:are by default eligible for Garbage Collector once the method completes

Note: 1) If an object doesn’t have any reference variable that object is always eligible for Garbage

Collection

2) **Even though object having the reference variable still there may be a chance of that object eligible for Garbage Collection** (Island of Isolation …Here ‘i’ is internal reference)

The methods to request JVM to run Garbage Collector:

**We can request JVM to run Garbage Collector** but there is **no guarantee** whether JVM accepts our request or not. We can do this by using the following ways.

**System.gc();**

Runtime.getRuntime().gc();

**finalization**:Just before destroying any object Garbage Collector always calls finalize() to perform clean up activities.

**Memory Leak**: memory leak in Java is a situation where some objects are not used by the application any more, but GC fails to recognize them as unused.

1**.referencing a heavy object** **with a static field.**public static final ArrayList<Double> list = new ArrayList<Double>(1000000);

here **ArrayList** shold not referenced by a **static variable**.Instead, it’s a local variable that gets created, used and then discarded:

2.Calling String.intern() on Long String:String str = new Scanner(new File("src/test/resources/large.txt"), "UTF-8").useDelimiter("\\A").next();

str.intern();

3.Unclosed Streams

try (BufferedReader br = new BufferedReader(new InputStreamReader(conn.getInputStream(), StandardCharsets.UTF\_8))) {// further implementation}

4.Unclosed Connections:URL url = new URL("ftp://speedtest.tele2.net");URLConnection urlc = url.openConnection();

5.Adding Objects with no hashCode() and equals() into a HashSet

when we start adding duplicate objects into a Set – this will only ever grow, instead of ignoring duplicates as it should. We also won’t be able to remove these objects, once added.

If the content is not changing frequently then we should go for the string.

If the content will change frequently and thread safety is required then we should go for StringBuffer.

If the content is changing frequently and thread safety is not required then we should go for StringBuilder

**ArrayList:** dynamic array

**add():**

1.checks if the array capable enough to add the new Element

2.if not then create a bigger array

3.copy all the existing elements to the bigger array.

4.add new Element to bigger array.

5. add new Element in the array.

transient Object[] eleData;

int size;

ArrayList(){

eleData={};

}

ArrayList(Collection e){

eleData=e.toArray();

size=eleData.length;

}

ArrayList(int siz)

{

if(siz>0){

eleData=new Object[siz];

}

else if(siz==0){

eleData=new Object[10];

}

}

void add(Object e){

ensureCapacity(size+1);

eleData[size++]=e;

return true;

}

public void add(int index, E element) {

ensureCapacityInternal(size + 1); // Increments modCount!!

System.arraycopy(elementData, index, elementData, index + 1,

size - index);

elementData[index] = element;

size++;

}

ensureCapacity(int siz){

if(eleData=={}){

siz=max(10,siz)

}

if(siz-size>0){

size=size\*3/2+1;

eleData=Arrys.copyOf(eleData,size);

}

}

Object remove(index){

Object x=eleData[index];

Arrays.copyOf(eleData,index+1,eleData,index,size-index-1)

return x;

}

**HashSet** uses HashMap:

1.Set internally uses **Hashmap**

2.**Set** **Element** will go as **key** to map

3.static **final object** will go as **value** to the map

4.**set returns false** if the element **already exists** in the map but internally map put replaces exiting value of the key.

5.**set returns true** if it a **new entry** in the map.

private static final Object PRESENT = new Object();

boolean set.add(E){

return map.put(E,PRESENT)==null;

}

boolean set.remove(E){

return map.remove(E) == PRESENT;

}

**HashMap: unique elements by key**

**Put():**

**1.** **Calculate hash code of the Key**

**2. Calculate index by using hash code.**

**3. Goes to the indexed bucket (contains a linked list)**

**4. Traverse the linked list and check if the map key is equals to the node key by using .equals method.**

**5. if there is any key match then existing node value will be replaced by the new given value for the given key.**

**6. if no key match till the end of the linked list then new node created with the given key & value.**

**Get():**

**1. Calculate hash code of the given Key**

**2. Calculate index by using hash code.**

**3. Goes to the indexed bucket (contains a linked list)**

**4. Traverse the linked list and check if the map key is equals to the node key by using .equals method.**

**5. if there is any key match then returns the value of the node.**

**6. if no key match till the end of the linked list then return null.**

**int hashCode = object.getHashCode();**

**Multiple objects can have same hashcode because hashCode is an integer so only integer range of hashcodes can be possible. So more than one object can have same hashcode.**

**Map.put(k1, v1);**

**Map.put(k2, v2);**

**Map.put(k1, v2);update**

**1.If both keys k1 and k2 has same hashcode then the index is also same then both points to same bucket linked list.**

**2.Traverse the linked list and check if the map key is equals to the node key by using .equals method.**

**3.if there is any key match then existing node value will be replaced by the new given value for the given key.**

**4. if no key match till the end of the linked list then new node created with the given key & value.**

**Why key comparison in Linked list:**

**1.If both keys k1 and k2 has same hashcode then the index is also same then both points to same bucket linked list.**

**2.All the nodes in the bucket Linked list has same hashCode, to distinguish between nodes we need key comparison to identify the node to replace or insert.**

static class Entry<K ,V> implements Map.Entry<K, V>

{

final K k;

V v;

Entry<K ,V> next;

final int hash; 1``

Entry(K k,V v,hash)

{

this.k=k;

this.v= v;

this.hash=hash;

}

}

Class HashMap<K,V>{

Entry[] table;

HashMap(int size){

table=new Entry[size];

}

}

static int indexFor(int h, int length) {

return h & (length-1);

}

public V put(K key, V value) {

if (key == null)

return putForNullKey(value);

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<K , V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

public V get(Object key) {

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<K , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

List:double linked list

WeakHasMap:week references as key .means if key object garbage collected then that map entry also garbage collected from map.maintain insertion order

entry of weak hashmap uses WeakRefrence

Identity HasMap:no hashcode,equals type for comparing object.It uses ==(reference equality check not logical),so faster.

uses IdentityHashCode(object)not hashCode

In hashMap keys to be immutable but in it dont require that constraint.

EnumSet:set of eum,maintain order

EnumMap:maintain order,null point if null key

copyOnWriteArraySet:impliments set interface and copyOnWriteArrayList so only unique elemnets ,insertion order and it is thread safe.

Priority Queue(not safe):put wil not block ,so outofmemory ,priorityBlockingQueue(thread safe) O(logn) comparable

ConcurrentLinkedQueue;put and take will not block,no null

SynchronousQueue:only one element can present in the queue,no null ,put ,take blocking methods

ArrayBlockingQueue(10):uses arry,FIFO,put,take block when full/empty

LinkedBlockingQueue:uses linked list

priorityBlockingQueue(comparator):put,take block

DelayQueue(comparator):type Delayed,getDelay(),compareto(),consumer can consume after given delay

ArrayDeque:both end,add==offer,peek,poll

LinkedBlockingDeQueue:

stack extends vector:E push,E pop,peek,empty

ConcurrentSkipListMap(comparator) impliments navigableMap:sorted based keys,desendingKeyset(),E floorEntry(k),Map headMap(k),E higherEntry(k)

ConcurrentSkipListSet(comparator)uses ConcurrentSkipListMap(add()==>put()):sorted like treeset,no null,highr(k),lower(k),tailSet(k)==navigable methods

Log Level Order:

OFF

FATAL

ERROR

WARN

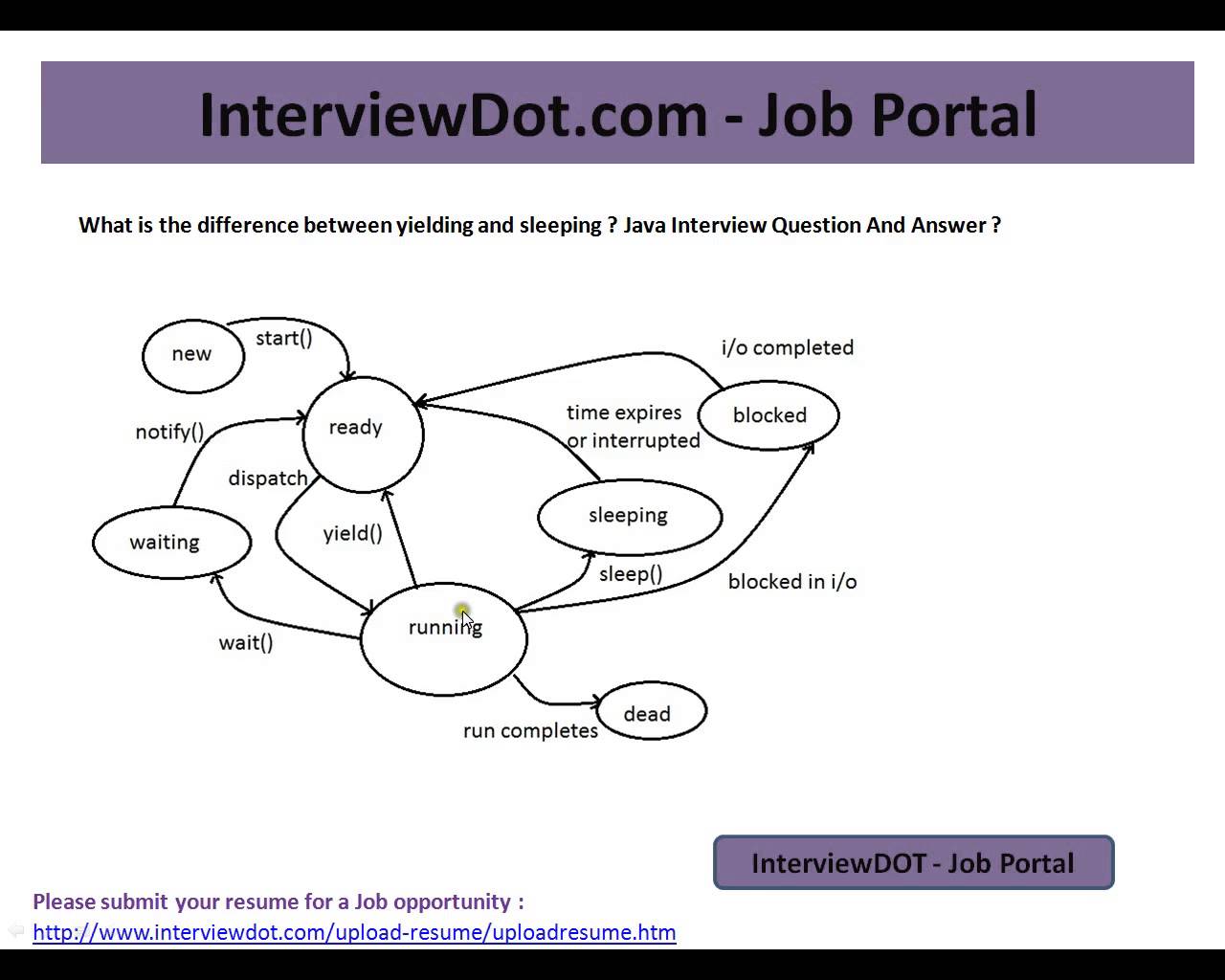
INFO

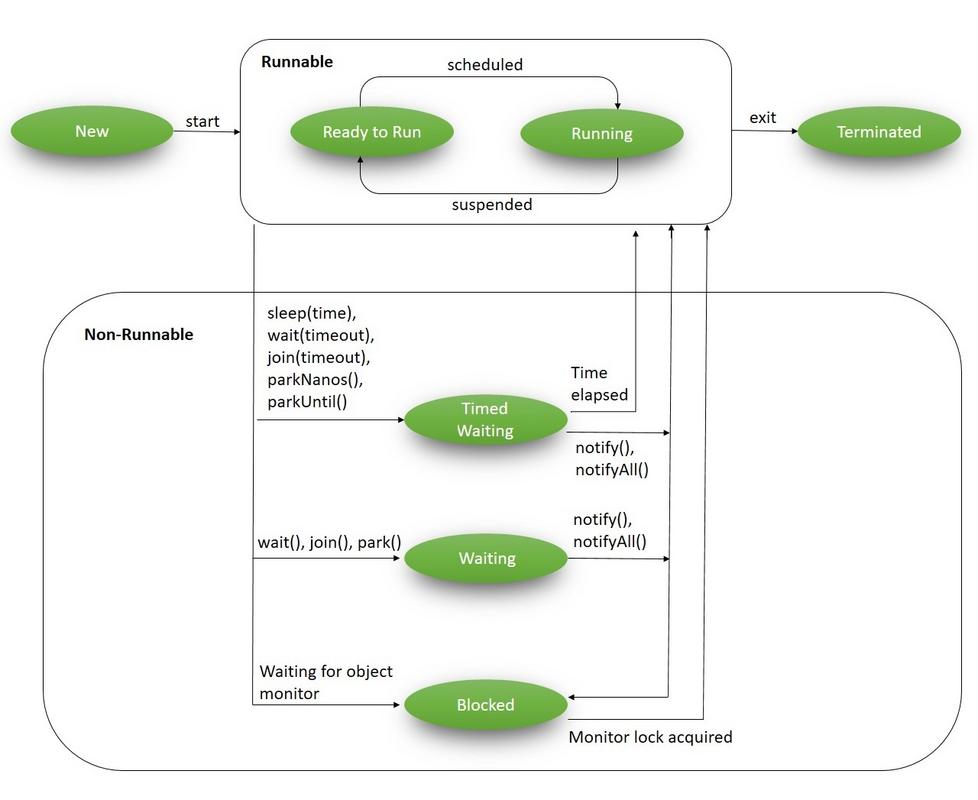
DEBUG

TRACE

ALL

**Threads**:





**lifecycle**:new-->start()-->runnable/ready-->wait(),join()-->waiting

-->runnable/ready-->wait-10/sleep(10),join(10)-->Timedwaiting

-->wait,acquireLock,ip/op,sync-->blocked

**Java concurrency**: only when multiple processors (CPU cores) available and it is for **best CPU utilization**

**Thread Scheduler ALG**: A component of Java that decides **which thread to run or execute** and **which thread to wait** is called a thread scheduler (ex: First Come First Serve, Time-slicing, Preemptive-Priority)

**Thread**: a thread is part of that running process.is a lightweight sub-process. New call stack gets created for each thread.

**Thread Scheduler**: A component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler (ex: First Come First Serve, Time-slicing, Preemptive-Priority)

1.Extending Thread Class: Thread itself implements Runnable and overrides with empty run method

2.Implementing a Runnable interface: task wont return anything. void run()

3.Implementing a Callable interface: task returns future object. future call()

**callable vs runnable:** future.get(10) is blocked statement.

public class GFG implements Runnable {

// method to start Thread

public void run()

{

System.out.println(

"Thread is Running Successfully");

}

public static void main(String[] args)

{

GFG g1 = new GFG();

// initializing Thread Object

Thread t1 = new Thread(g1);

t1.start();

}

}

**Multithreading** : running multiple threads simultaneously/Parallelly.

**Can we start a thread twice**?: no==illigal thread state exception

**What if we call Java run() method directly instead start() method?:** run() is normal method call.

**join**() : **Main** thread **waits** until the joined thread gets completed.

**Thread.Yield():** **running thread pauses and gives chance to other thread.** Pause the execution of current thread and gives control to other higher priority thread,if not same thread will be called again.No guarenty run

**Synchronization**: **only one thread** isallowed to access shared resource **at the same time**, to avoid data inconsistence of a shared resource in multi-threaded environment

If a method or block declares as synchronized then at a time only one thread allowed to execute that method or block on the given object

🡪 Don’t use synchronization in a single threaded env.

Synchronization is built around an internal entity known as the **lock or monitor**. **Every object has a lock** associated with it. By convention, **a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them**, and then release the lock when it's done with them.

**Monitor vs lock**: each object associated with a monitor which a thread can lock or unlock.only one thread at a time may hold a lock on a monitor.any other thread attempting to lock that monitor are blocked until they can obtain the lock on the monitor.

**Monitor**: Monitor is a synchronization construct that allows threads to have both mutual exclusion (using locks) and cooperation i.e. the ability to make threads wait for certain condition to be true (using wait-set).

**lock**: is kind of data which is logically part of an objects header on the heap.

locks help threads to work independently on shared data without interfering with one another, wait-sets help threads to cooperate with one another to work together towards a common goal e.g. all waiting threads will be moved to this wait-set and all will be notified once lock is released.

**Concurrency vs Parallelism:**

**concurrency**:threads will look like simultanious but not actually(cpu shared between threads when idle)

**Parallelism**:same time using multi-core infrastructure of CPU, by assigning one core to each task or sub-task.

**In single core CPU, you may get concurrency but NOT parallelism.**

**Thread Synchronization are of two types:**

1.**Mutual Exclusive:**

1.Synchronized method.

2.Synchronized block.

3.Static synchronization.

2.**Inter-Thread Communication:**

InterThread communication using

1.wait,

2.notify,

3.notifyAll

**Producer and consumer:**

1.synchronized block & wait and notify using a Queue(add(),poll()) using one Queue(can be done using 2 objects(full&empty) and a queue)

2.using a BlockingQueue(put(Object),Object=take()) BlockingQueue<Integer> queue=new ArrayBlockingQueue<>(5);

3.producerSemaphore=new Semaphore(1),consumerSemaphore =new Sempahore(0)(acquire(),release()),MyQueue(put(producerSemaphore.acquire(),consumerSemaphore.rlease()),get() our methods and our class)

4.Reentrant and Condition==>lock -->if full -->fullCondition.await() -->else add into queue -->emptyCondition.signalAll()-->unLock()

**InterThread communication** using wait ,notify with a single shared Queue<Integer> queue= new LinkedList<>();

while(true){

synchronized(queue){xc

while(queue.size==5)//queue.empty==true

queue.wait();

queue.add(item)//queue.poll();

queue.notify();

}

}

if size full then producer goes to waiting and if empty then consumer goes to waiting

**Producer**:

run()

{

while(true){

synchronized(queue){

while(full){queue.wait();}

queue.add();

queue.notify();

}

}}

**Consumer**:

run(){

while(true){

synchronized(queue){

while(empty){queue.wait()}

queue.poll();

queue.notify();

}

}

}

**wait** **vs** **sleep**:

releases lock --- not realesae lock when synchronized

Object class method --- Thread clas method

non static --- static

wait should be notified by notify --- after specified amount of time

wait called from syncronized block --- any where

wait called from loop --- no such constraint

**Why wait always called inside a synchronized method and in while:**

class **BlockingQueue** {

Queue<String> buffer = new LinkedList<String>();

public void put(String data) {

while(buffer,isFull())

wait();

buffer.add(data);

notify(); // Since someone may be waiting in take!

}

public String take() throws InterruptedException {

while (buffer.isEmpty()) // don't use "if" due to spurious wakeups.

wait();

return buffer.remove();notify();

}

}

**This is what could potentially happen:**

A consumer thread calls take() and sees that the buffer.isEmpty().

Before the consumer thread goes on to call wait(), a producer thread comes along and invokes a full give(), that is, buffer.add(data); notify();

The consumer thread will now call wait() (and miss the notify() that was just called).

If unlucky, the producer thread won't produce more give() as a result of the fact that the consumer thread never wakes up, and we have a **dead-lock**

**solution is obvious:** Use synchronized to make sure **notify is never called between isEmpty and wait.**

**can be solved by blocking queue:**.put(e)/take() --waits untill queue has some space--no need wait,notify

**cab be solved by semaphore**:producer=semaphore(1),consumer=semaphore(0)

put(){

producer.acquire();

item;

consumer.release();

}

get(){

consumer.acquire();

item

producer.release();

}

**why synchronization**:count++/ a[0]++==>1.copy value from ram to regi.2.increment value in register.3.update ram by the incremented value

so if increment thread updated data in reg but not in ram then dec thread takes old value and dec and copy to ram the val=1 not 0

both threads can't do synchromnized increament and decreament at a time like thread-A increament() and thread-B decreament()

if a thread executing a sync-method then no other thread is allowed to execute any other sync-method but can access un-sync.

**Join with sync-method:** count++==>count=count+1 so join alone cant get final result also need sync-method in run method(count++);

**Drawback of synch**: performance reduced(use only when shared resource in multi-thread).so for single threaded use arralist over vector

inproper sync cause deadlock.

method is synchronized ,other threads has to wait for complte method,so sync block is better

use synchronization to avoid data inconsistence of a shared resource in multi threaded environment

dont use synchronization in a single threaded env.

1.multiple threads can access the non-sync-method concurrently.

2.sleep holds the lock where as wait release the lock

3.thread can hold multiple locks at a time

4.a thread can call any other syncronizred methods of the same object using the lock that already holds.locks are reentrant

5.static methods can be syncronized.

6.2 threads can not call 2 different static synchronized methods of the same class.but can call instatic(sync,non-synch) ,static non-sysnch

7.sleep,yield,join does not release the lock where as wait,notity,notifyAll release

8.why wait,notify in object:beacuse lock is related to object not thread

9.syncronizred method can lock the method object.syncronizred block can pass any object

10.run method can be syncronized but one thread start after the prevoius thread ends.

11.invalid monitor state exception if wait notify not called in sync-method

12.syncronizred to another synchronized method possible and also uses the earlier lock to access

**Synchronized is for blocks and methods, volatile and atomic is for variables.**

**If a variable value keeps on changing, then we go with atomic or volatile variables.**

**Volatile**==**AtomicBoolean**: The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory.

volatile is enough only when there is one writer thread and remaining all are reader thread,but if we have multiple writes then volatile+syncronized

**HappensBefore** :once reading of voltile variable done then all the earlier changed variables(non-volatile) values updated.

**Atomic Variables**:intead of i++;use new AutomicInteger(1).increment()/incrementAndGet()/decrementAndGet()/addAndGet()/compareAndSet(expected,newValue)

volatile for flags:Volatile==AtomicBoolean

AtomicInteger/AtomicLong for counters -->needs lot of synchronizations on every increment use LongAdder or LongAccumulator

**LongAdder(1.8) :has more scalable strategy to reduce contention(when resources are not readily available) and improve performance than Atomic variables**

**LongAdder**:increment(),sum():every thread will work only on local cache copy and finally .sum() will combine all the thread results, so only synchronization at sum.

**LongAccumulator**((x,y)->x+y,0):accumulate(1)/get() -->any operartion not like Adder(only add and delete)

**cpu cores** --(process threads)--> registers-> L1 cache/core->L2 cache/core-> L3 cache(shared cache)->primary memory(RAM)--> secondary memory(disk)

**SynchronusQeue**:can hold only 1 task ,so in cachedThreadPool new thread created for assigning the submitted task immidiatly.

**Exchanger**:same as synchronus queue but hand off in both the directions.exchanges tasks between threads.empty buffer to produce and full buffer to consumer

**Reactive Framework**:if we there are more IO intensive oprations then we normally go for chached threadpool but it take lot of queue memory to hold the blocked thread.

So in reactive we do invoke and mension callback functionality as well,we dont need to wait any where.

ex:Flux

**Java Fibers**(coroutines)(very light weight) solve Flux complicated api problem.by mounting and unmouting the tasks.

when an IO, instead of waiting in blocked state thread removes the task and saves task and its state and take any other task.if any other thread is free then this saved task is resume from the previous step.

**SpinLock**:keep trying to acquire the lock without going to wait state(beacuse of context switching is costly) ,this thread will run on other cpu core.

assuming that most locks are used only for short period of time.

if a task take more time then you are wasting cpu core by keep on trying for lock and not allowing other thread to run on the core,so can cuase starvation.

**starvation**:If a thread is not granted CPU time because other threads grab it all, it is called"starvation". The thread is "starved to death"because other threads are allowed the CPU time instead of it.The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute

**Striped Lock**:single lock for group of items Striped.lock(10).get(bag.id).lock/unlock==>gets the lock for the bag and do normal lock and unlock.

for 200 bags only 10 locks==>so per 20 bags only 1 lock.(lock=hashcode%totalLocks)

**Stop the task if it exceeds 10 mins:**

1.stop a thread:using interrupts: with the help of t1.interrupt()/future.cancel(true)(for callable)/shoutDownNow()(for executers)/volatile/atomicBoolean .in run method we keep on check while(Thread.currentThread.isInterrupted()/false/false)

2.after some time:thread1.sleep(10\*1000) then call interrupt()/volatileVariable(false)/atomicBoolanVariable(false);

ScheduledThreadPool().shedule(()->step:1,10\*1000);or Future f=Exectors.newFixedThreadPool(5).submit(new Task());f.get(10\*1000) control goes to timeout catch then do step:1 in catch

**Get all thread results for main thread:**

1.results of all the threads:

2.only results of the threads which can get within 10 mins:

**sol-1:**Thread.sleep(10) in main is not proper bcz if all the threads complted with in 1 min then 9 minutes main thread has to wait unncessqarly.

if not sleep then main thread will resturn with empty values.

**sol-2:**Best solution is CountdownLatch(await(10) in main and countDown() in run)

**sol-3:**CompletableFuture:CompletableFuture.allOf(task1,2,3).get(10); then take results

also with Future/Conditions/Phaser

**Deadlock:** Thread A waits for the resource which held by thread B and thread B waits for the resource which held by thread A. **each Thread is holding a resource and waiting for another resource acquired by some other process.**

when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

**Deadlock with single thread(main) thread**:Thread.currentThread().join() , wont execute main method next statement.

**DeadLockDetection:**In JvisualVM we can see and jps -l -m==gets serviceId then jstack serviceId--thread dump

**DeadlockPrevention and Avoid:**

1.accessing of resources should be ordered

2.avoid nested locks:avoid locking other resource if you already have one.it is almost impoosible with only one object lock

3.avoid un-necessary locks

4.join(maxTime) not .join()

5.acquired/release

6.ReentrantLock.tryLock(100)

**ExecutorsFramework**: (**Java Thread Pool**): Better performance It saves time because there is no need to create a new thread.

helps in thread Creation,management,task submition and execution

1.ExecutorService s=Executors.newSingleThreadedExecutor();only one thread created and available in the pool to handle the task.

2.Executors.newFixedThreadPlool(2); given number of threads created in the pool to handle the tasks

3.Executors.scheduledThreadPool(2);.schude(),.scheduleAtFixedRate(),fixedDelay

4.Exrcutors.newCachedThreadPool(); new thread created for each task if there is no thread available to handle the task

5.ForkJoinThreadPool();

Future<String> result = executorService.submit(callableOneTask);

List<Future<String>> futures = executorService.invokeAll(myCallableListOfTasks));

**ThreadPoolExecutor**: Executors class provides simple implementation of ExecutorService using ThreadPoolExecutor, We can specify the number of threads that will be **alive** when we create ThreadPoolExecutor instance, and we can limit the size of the thread pool and create our RejectedExecutionHandler implementation to handle the jobs that can't fit in the worker queue.

**CompletableFuture**: To configure the tasks and **run chain of tasks one after completing the other**, is to do asynchronous calls without stopping main thread(can be stoped by .get(10)).if want to start a task after other task then you have a chance to configure the steps

completableFuture.**supplyAsync**(()->function(object)).**thenApplyAsync**((1stResult->())).**thenAcceptAsync**((2ndndRslt)->());/**runAsync**(new Task())

your main thread does not block/wait for the completion of the task and it can execute other tasks in parallel.

List<CompletableFuture<String>> completableFutures = Arrays.asList(completableFuture1, completableFuture2);

CompletableFuture<Void> resultantCf = CompletableFuture.**allOf**(completableFutures.toArray(new CompletableFuture[completableFutures.size()]));

CompletableFuture<List<String>> allFutureResults = resultantCf.thenApply(t -> completableFutures.stream().map(CompletableFuture::join).collect(Collectors.toList()));

System.out.println("Result - " + allFutureResults.get());

**ForkJoinPool** implimented based on WorkStealingAlgorithm:used hen one tas can be dided into subtasks and further sub tasks and joined all together to get consolidated output

Each thread has its own deque ,thread keep picking from from its own queue.if other thread dont have any task even in local and common queue the it steals the task from back of 1st thread queue.

task T dividend into 12 subtasks and 4 process with 3 sub-tasks.if p3 is busy and p2 is free then p2 will steal work of p3

Fork():task divided into subtasks

join():aggrigation of subtask results

ForkJoinPool f=new ForkJoinPool(paeallism level:number of processors)

List<Future<Integer> l=f.invokeAll(taskList);then extends RucursiveTask<Object> then override object compute(){t1=new RucursiveTask();t1.fork();t1.join() }

==workStealingPool()

==fixedThreadPool()

**Semaphores** are also a way to synchronize threads. A semaphore is created with a certain value for its counter

semaphore is a counter (integer) that allows a thread to get into a critical region if the value of the counter is greater than 0

the counter is decremented by one. Otherwise, the thread waits until it can go,

when the thread leaves the critical region, the counter is incremented by one .

Semaphore s=new Semaphore(10);

s.acquire():p; max 10 threads can run on the method,decreaced count by 1

s.release():v;all acquired lock willl be release

A mutex is a "Mutual Exclusion Semaphore".only 1 thread can enter

sem mutex = new sem(1)

P(mutex)

//Critical region

V(mutex)

ex:database connections or a data pool.

semaphores are a powerful way to solve concurrency problems,

If you only need mutual exclusion, synchronized blocks are a better solution.

The problems with semaphores is that you can forget to call the release method and this can cause deadlocks that are sometimes difficult to find

**Mutex** is basically mutual exclusion. Only one thread can acquire the resource at once. When one thread acquires the resource, no other thread is allowed to acquire the resource until the thread owning the resource releases. All threads waiting for acquiring resource would be blocked.

Semaphore is used to control the number of threads executing. There will be fixed set of resources. The resource count will gets decremented every time when a thread owns the same. When the semaphore count reaches 0 then no other threads are allowed to acquire the resource. The threads get blocked till other threads owning resource releases.

In short, the main difference is how many threads are allowed to acquire the resource at once ?

•**Mutex** --its ONE.

•**Semaphore** -- its DEFINED\_COUNT,

**CyclicBarrier**(size) :number of threads want to wait for each other at common point(called barrier)

each thread calls int c=b.await():decreemnt by 1 size,when last calls await() it has reached barrier then all threads start processing again

Thread t1=new Thread(cyclicBarrier);Thread t2=new Thread(cyclicBarrier);

if(c==0)

use can resuse cyclic barrier for other size set of threads

similar to CountDownLatch ,it calls await() instead of countdown();

you can use CyclicBarrier instead of CountDownLatch but opposite is not possible beacuse you can not reuse the latch once count decreased to 0

**CountDownLatch**(size) :

c.countdown();decremented the size by 1

when countdown=0 in run then c.await() method notified and next statemts executed

**Phaser**(size):can be represnted as CountDownLatch(AwaitAdvance()=await(),arrive()=countdown()) and cyclicBarrier(arriveAndAwaitAdvance()=await())

and register/deregister any new task at any time and bulkRegister(size)

**join can not used instead of countdown:**

1.when executer service used to create thread instead of thread class

2.we can make nextStatemets to wait till some % task but in join we can't

when t1,t2 ,t3 then we can make t3 to execute after t1 and t2 completed using CountDownLatch

when t1,t2,t3 then we can await all the thread at a point till the barrrier broken by making barrier count =0

In CountDownLatch can't be re-usable

In cyclicBarrier can be re-usable

=============================================================================

**CountDownLatch**:

This class enables a java thread to wait until other set of threads completes their % of tasks.

CountDownLatch.countDown(); Each invocation of method decreases the initial count set in constructor, by 1.

So,when all N threads have call this method, count reaches to zero, and main thread is allowed to resume its execution past await() method.

Calling await() blocks the thread until the count reaches zero.

EX:public class CountDownLatchDemo

{

public static void main(String args[])

throws InterruptedException

{

CountDownLatch latch = new CountDownLatch(4);

Worker first = new Worker(1000, latch,"WORKER-1");

Worker second = new Worker(2000, latch,"WORKER-2");

Worker third = new Worker(3000, latch,"WORKER-3");

Worker fourth = new Worker(4000, latch,"WORKER-4");

first.start();

second.start();

third.start();

fourth.start();

// The main task waits for four threads

latch.await();

// Main thread has started

System.out.println(Thread.currentThread().getName() +

" has finished");

}

}

class Worker extends Thread

{

private int delay;

private CountDownLatch latch;

public Worker(int delay, CountDownLatch latch,

String name)

{

super(name);

this.delay = delay;

this.latch = latch;

}

@Override

public void run()

{

try

{

Thread.sleep(delay);

System.out.println("aaa");

latch.countDown();

System.out.println(Thread.currentThread().getName()

+ " finished");

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

Output:

aaa

WORKER-1 has finished

aaa

WORKER-2 has finished

aaa

WORKER-3 has finished

aaa

WORKER-4 has finished

main has finished

**CyclicBarrier**:when a number of threads (also known as parties) want to wait for each other at a common point, also known as the barrier before starting processing again.

Its similar to CountDownLatch but instead of calling countDown() each thread calls await() and when last thread calls await() which signals that it has reached the barrier, all thread started processing again, also known as a barrier is broken.

You can use CyclicBarrier wherever you want to use CountDownLatch, but the opposite is not possible because you can not reuse the latch once the count reaches to zero

a CyclicBarrier is used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output.

The CyclicBarrier can also be initialized with some action that is performed once all the threads have reached the barrier. This action can combine/utilize the result of computation of individual thread waiting in the barrier

You can still reuse the barrier object and if a thread calls barrier.await() again, it will wait for four worker thread before it gets wake up call. By the way, If barrier is broken before a thread calls await() then this method will throw BrokenBarrierException.

public class HelloHP {

public static void main(String args[]) throws InterruptedException, BrokenBarrierException {

CyclicBarrier barrier = new CyclicBarrier(4);

Party first = new Party(1000, barrier, "PARTY-1");

Party second = new Party(2000, barrier, "PARTY-2");

Party third = new Party(3000, barrier, "PARTY-3");

Party fourth = new Party(4000, barrier, "PARTY-4");

Party first1 = new Party(1000, barrier, "PARTY-11");

Party second1 = new Party(2000, barrier, "PARTY-21");

Party third1 = new Party(3000, barrier, "PARTY-31");

Party fourth1 = new Party(4000, barrier, "PARTY-41");

first.start();

second.start();

third.start();

fourth.start();

first1.start();

second1.start();

third1.start();

fourth1.start();

System.out.println(Thread.currentThread().getName() + " has finished");

}

}

class Party extends Thread {

private int duration;

private CyclicBarrier barrier;

public Party(int duration, CyclicBarrier barrier, String name) {

super(name);

this.duration = duration;

this.barrier = barrier;

}

@Override

public void run() {

try {

Thread.sleep(duration);

System.out.println(Thread.currentThread().getName() + " is calling await()");

barrier.await();

System.out.println(Thread.currentThread().getName() + " has started running again");

} catch (InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

}

main has finished

PARTY-11 is calling await()

PARTY-1 is calling await()

PARTY-2 is calling await()

PARTY-21 is calling await()

PARTY-11 has started running again

PARTY-2 has started running again

PARTY-1 has started running again

PARTY-21 has started running again

PARTY-31 is calling await()

PARTY-3 is calling await()

PARTY-4 is calling await()

PARTY-41 is calling await()

PARTY-4 has started running again

PARTY-41 has started running again

PARTY-3 has started running again

PARTY-31 has started running again

==================================================================

**await**() is a blocking call and it blocks until count reaches zero.

==================================================================

**join**(): It will put the current thread on wait until the thread on which it is called is completed/dead. If thread is interrupted then it will throw InterruptedException.

**Join** vs **CountDownLatch**

1.with join(), thread can be unblocked only when joined thread has finished its execution while in CountDownLatch a thread can decrease the count anytime either on completion of thread or in between based on any condition.

This way we can get better control over unblocking of the thread instead of solely depending on the completion of joined thread.

2.While using ExecutorServices,ExecutorServices do not directly reveal how they are executing tasks, so you would have to use a CountDownLatch:

3.Let's assume you want to start a game when at least 2 players should be available. You can use countdownlatch in this case. But you can't achieve this using join easily because you don't have another thread(player in this case) on which you can write join().

**Semaphore**:given number of threads can enter into the critical section at a time.Sempaphore s=new Semaphore(10,fair);s.acquire()/release()

**Mutex**:only one thread at a can enter into the critical section at a time. Sempaphore s=new Semaphore(1);//better to use synchronized block because it requires release() to acuired lock if not deadlock

**Reentrant Lock(**syncronized+fairness(longest waiting thread gets chance)):ReentrantLock allows threads to enter into lock on a resource more than once before unlocked. When the thread first enters into lock, a hold count is set to one.

Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one.

For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

**Reentrant vs synchronized**:which provides more control on lock

you don't need to use a block structure for locking and it can even hold a lock across methods(create in 1 method and release in other method)

1) Ability to lock interruptibly.

2) Ability to timeout while waiting for lock.(tryLock(1000))

3) Power to create fair lock(longest waiting thread gets chance).

4) API to get list of waiting thread for lock.

5) Flexibility to tryForLock without blocking.

**ReadWriteLock**:parts of locks,read acces:readLock,and write access:writeLock,

Read lock may held by Multiple threds simultaniously but only 1 write lock

prefered when Fewer writes than reads

rLock.readLock().lock()/unlock();

rLOck.writeLock().lock()/unlock();

Thread 1 gets read access.

Thread 2 requests write access but is blocked because there is one reader.

Thread 1 re-requests read access (re-enters the lock), but is blocked because there is a write request

In ReentrantLock only one thread at a time but in ReadWriteLock any no of reader or 1 writer thread at a time.

Condition:reEntrantLock.newCondition().await()/signalAll()

while(full)

fillCondition.await()

push()

emtyContion.signalALL();

===

while(empt)

emtyContion.await();

fullcodition.signalALL();

**StampedLock**:same like reentrantReadWrie ,allow optimistic locking for read operartions but not reetrant support,used when more reads than writers

**Commonly used methods of Thread class:**

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted(): tests if the current thread has been interrupted.**