Contents

[Read these webPages 5](#_Toc41417085)

[Topics to Learn 6](#_Toc41417086)

[Git 8](#_Toc41417087)

[Branch 8](#_Toc41417088)

[Question 10](#_Toc41417089)

[Puzzle 11](#_Toc41417090)

[Cool Concepts 12](#_Toc41417091)

[Java 13](#_Toc41417092)

[Software Architecture 15](#_Toc41417093)

[Java Version 16](#_Toc41417094)

[Definition 18](#_Toc41417095)

[Concepts 22](#_Toc41417096)

[Enhanced for Loop 22](#_Toc41417097)

[lambda expression 22](#_Toc41417098)

[Stack/Heap 23](#_Toc41417099)

[Strings 23](#_Toc41417100)

[Deep, Shallow, Lazy, Clone 23](#_Toc41417101)

[Immutable 23](#_Toc41417102)

[Exception 24](#_Toc41417103)

[Enhanced try block 24](#_Toc41417104)

[Reflection 25](#_Toc41417105)

[Enumeration 25](#_Toc41417106)

[Serialization 25](#_Toc41417107)

[Pass by value or reference 26](#_Toc41417108)

[Reference 26](#_Toc41417109)

[Interface 26](#_Toc41417110)

[Default Methods 26](#_Toc41417111)

[Functional Interface 26](#_Toc41417112)

[Garbage Collection 27](#_Toc41417113)

[Types of Garbage Collectors 27](#_Toc41417114)

[Serial GC 28](#_Toc41417115)

[Parallel GC 28](#_Toc41417116)

[Concurrent Mark & Sweep (CMS) GC 28](#_Toc41417117)

[Garbage First GC 28](#_Toc41417118)

[Data Structures 29](#_Toc41417119)

[Binary Tree 29](#_Toc41417120)

[Linked List 29](#_Toc41417121)

[Sorting 30](#_Toc41417122)

[HeapSort 30](#_Toc41417123)

[Collection 31](#_Toc41417124)

[Fail Safe 31](#_Toc41417125)

[fail fast 31](#_Toc41417126)

[Iterable 31](#_Toc41417127)

[Iterator 31](#_Toc41417128)

[Collections 31](#_Toc41417129)

[Vector 31](#_Toc41417130)

[ArrayList 32](#_Toc41417131)

[LinkedList 32](#_Toc41417132)

[Priority Queue 32](#_Toc41417133)

[Map 32](#_Toc41417134)

[HashMap 33](#_Toc41417135)

[LinkedHashMap 34](#_Toc41417136)

[Hashtable 34](#_Toc41417137)

[Set 34](#_Toc41417138)

[HashSet 34](#_Toc41417139)

[LinkedHashMap 34](#_Toc41417140)

[Complexity 35](#_Toc41417141)

[Standard Algorithms 35](#_Toc41417142)

[Order of Complexity 35](#_Toc41417143)

[Design Pattern 37](#_Toc41417144)

[Behavioural 37](#_Toc41417145)

[Chain of Responsibility 37](#_Toc41417146)

[Strategy 37](#_Toc41417147)

[Observer 38](#_Toc41417148)

[Command Pattern 38](#_Toc41417149)

[Iterator Pattern 39](#_Toc41417150)

[Template Method 41](#_Toc41417151)

[State Pattern 41](#_Toc41417152)

[Structural 42](#_Toc41417153)

[Decorator Pattern 42](#_Toc41417154)

[Adapter Pattern 43](#_Toc41417155)

[Bridge Pattern 44](#_Toc41417156)

[Façade 45](#_Toc41417157)

[Proxy Pattern 45](#_Toc41417158)

[Composite 46](#_Toc41417159)

[FlyWeight 46](#_Toc41417160)

[Creational 46](#_Toc41417161)

[Static Factory 46](#_Toc41417162)

[Factory Method 46](#_Toc41417163)

[Abstract Factory 47](#_Toc41417164)

[Singleton 47](#_Toc41417165)

[Builder Pattern 49](#_Toc41417166)

[Prototype 50](#_Toc41417167)

[Design Principles 51](#_Toc41417168)

[Single responsibility 51](#_Toc41417169)

[Open-Closed Principle 51](#_Toc41417170)

[Liskov Substitution Principle 52](#_Toc41417171)

[Interface Segregation Principle 52](#_Toc41417172)

[Dependency Inversion Principle 52](#_Toc41417173)

[Threads 53](#_Toc41417174)

[Simple Definitions 53](#_Toc41417175)

[Thread Safety 54](#_Toc41417176)

[Volatile 54](#_Toc41417177)

[synchronized block 54](#_Toc41417178)

[Spin Lock 57](#_Toc41417179)

[Read Write Lock 57](#_Toc41417180)

[Blocking Queue 57](#_Toc41417181)

[Thread Pools 58](#_Toc41417182)

[Non-blocking algorithms 58](#_Toc41417183)

[Future Object 58](#_Toc41417184)

# Read these webPages

http://www.crazyforcode.com/design-patterns/

<https://stackoverflow.com/questions/61629383/what-is-encapsulation-in-the-real-world-really>

<https://stackoverflow.com/questions/16635398/java-8-iterable-foreach-vs-foreach-loop>

**Collection:**

https://stackoverflow.com/questions/28373729/why-synchronize-a-synchronized-list

https://stackoverflow.com/questions/9468187/collections-synchronizedlist-and-synchronized

https://stackoverflow.com/questions/14932034/in-java-vector-and-collections-synchronizedlist-are-all-synchronized-whats-th

https://stackoverflow.com/questions/13151166/collections-synchronizedmapnew-linkedhashmap-is-not-making-map-threadsafe

https://stackoverflow.com/questions/510632/whats-the-difference-between-concurrenthashmap-and-collections-synchronizedmap

https://web.archive.org/web/20140604083201/http://www.codercorp.com/blog/java/why-concurrenthashmap-is-better-than-hashtable-and-just-as-good-hashmap.html

https://www.ibm.com/developerworks/java/library/j-jtp08223/j-jtp08223-pdf.pdf

<https://stackoverflow.com/questions/61629383/what-is-encapsulation-in-the-real-world-really>

**Streams**:

https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

# Topics to Learn

|  |  |
| --- | --- |
| Java Concepts | ~~Lambda and method references~~ |
| Streams and its Methods |
| ~~Enum~~ |
| ~~Inner Class~~ |
| ~~Anonymous Class~~ |
|  | Class Diagram / UML |
| Exception Handling |
| ~~IO and Serialization~~ |
| Complexity |
| Regular Expressions |
| ~~Collections~~ & Generics |
| Sorting |  |
| Searching |  |
| Data Structure | ~~List~~ |
| ~~Stack~~ |
| ~~Queue~~ |
| Tree |
| Map |
| Threading | ~~Basics~~ |
| Executor Framework |
| Design Patterns |  |
| Design Principles |  |
| Java System | Java Architecture |
| Class Loader and Boot Strap |
| Java Virtual Machine |
| Memory | ~~stack/heap/Ram/CPU Cache/CPU registers~~ |
| Java Memory Management |
| ~~Garbage Collection~~ |
| General Algorithms | 2 pointer algorithm 3 pointer algorithm Sliding Window algorithm |
| Build Management | ANT/Maven |
| GitHub |
| CI tools |
| Tests | Hacker Rank |

Difference between Liskov Substitution principle and Interface Segregation

<https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

https://docs.oracle.com/javase/8/docs/technotes/tools/findingclasses.html#bootclass

# Topic Finisher

Read these blogs or watch the video once a topic is finished. It would be refresher/settle the topic in mind

1. Check Jenkov
2. <https://www.youtube.com/c/DefogTech/playlists>

# Git

 A shell is a terminal application used to interface with an operating system through written commands.

Bash is a popular default shell on Linux and macOS.

Git Bash is a package that installs Bash, some common bash utilities, and Git on a Windows operating system.

GitBash is a command line interface for Git which works based on linux commands

GitCMD is a command line interface for Git which works based on Windows commands.

GitGUI, provides UI.

git config --global user.name "JamesTharakan"

git config --global user.email "jamestharakan85@yahoo.com"

git clone <https://github.com/JamesTharakan/cognitiveLearning.git>

*git config --global --edit*

Create a folder and run **git init.** This directory will become a repository. Add files to it.

Create a repository in github.com and link these two by using :

git remote add origin https://github.com/<name>/<repoName>.git

## Branch

$ git branch -d branch\_name //delete

$ git branch -D branch\_name

$ git push origin --delete <branch\_name>

deletes the remote branch

Revisit

https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes

Showing Your Remotes

$ git remote

origin

To see which remote servers you have configured, you can run the git remote command. It lists the short names of each remote handle you have specified.

If you have cloned your repository, you should at least see origin, that is the default name Git gives to the server you cloned from.

$ git remote -v

origin https://github.com/JamesTharakan/cognitiveLearning.git (fetch)

origin https://github.com/JamesTharakan/cognitiveLearning.git (push)

shows you the URLs that Git has stored for the short name(origin) to be used when reading and writing to that remote

First, you need to create your branch locally

git checkout -b your\_branch

After that, you can work locally in your branch, when you are ready to share the branch, push it.

The next command pushes the branch to the remote repository origin and tracks it.

git push -origin your\_branch

Your Teammates/colleagues can push to your branch by doing commits and then push explicitly

... work ...

git commit

... work ...

git commit

git push origin HEAD:your\_branch

git reset HEAD <filePath>

git log --branches --not --remotes=origin

Shows all commits that are in any of local branches but not in any of remote-tracking branches for origin

(What you have that origin does not).

git checkout -b branchname origin/branchname

Here, by default we are setting the upstream branch, so you will not be facing the mentioned issue.

If we push the changes to a <branch> using "-u" , then all your future pushes will be done to that <branch>

# Question

1. Sort method in collection

2. add() addAll()

3. remove() removeALL()

4. Retain() clear()

5. NavigationSet, NavigationMap

Collection col = new HashSet() is **better** than HashSet set = new Hashset() because we should code to the interface.

But what if, I want to use the methods that are declared and defined in HashSet

Answer: we cannot. So, use type Cast works. But type cast is not a good option as it is creating overhead of type casting. However, Java has fast and good type casting process.

**Questions**:

* Why Type Casting is bad ?
* Why is null check bad? Cost of null check is more?

Is there a difference between null != someThing **and** someThing != null

* JavaBean and similarities with the Builder Design pattern
* "Why to Sync a synchronized list?

<https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedList(java.util.List)>:

~~In the above API, an arrayList is created after synchronizing an array. We then synchronize the new created arrayList before/while using it.~~

# Puzzle

* Recursive function to do substring search
* Find the missing card in a deck
* Build Amazon filters, builder pattern
* Create an Iterator to iterate custom objects
  + Done. designPatterns.behavioural.iterator.MyIterator.java

# Cool Concepts

**Branch Prediction:**

"Processing a sorted array id faster than unSorted.

https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array/11227902#11227902"

# Java

**Robust and Secure**

There are no explicit programmer-defined pointer data types, no pointer arithmetic, and automatic garbage collection.

**Architecture Neutral**

Generates bytecodes--an architecture neutral intermediate format designed to transport code efficiently to multiple hardware and software platforms.

**Portable**

Specifies the sizes of its basic data types and the behaviour of its arithmetic operators.

Those source files are compiled into .class files by the javac compiler. A dot class file does not contain code that is native to your processor; it instead contains bytecodes — the machine language of the Java Virtual Machine. The java launcher tool then runs your application with an instance of the Java Virtual Machine.

**Platform**

A platform is the hardware or software environment in which a program runs. Most popular platforms like Windows, Linux, Solaris OS, and Mac OS. Most platforms can be described as a combination of the operating system and underlying hardware.

The Java platform differs from most other platforms, ~~in that~~ it is a software-only platform that runs on top of other hardware-based platforms.

myProgram.java -> API ->JVM -> Hardware

**Compilation**

Java code is converted into byte code by the javaC. The byte code (dot class) is the converted to Machine instruction.

The interpreter, one of the JVM components, uses the dictionary to convert byte code instruction to machine language, One line at a time.

JVM keeps performance counter to keep track of each snippet/method. Once the counter reaches threshold, it uses the c1 compiler.

The c1 compiler, optimises the code and cache the complied code in code cache (Very small. 240MB) in the JVM.

Further, JVM does code profiling to find hottest spot in the code. Then c2 compiler is used. Which performs heavier optimization. Uses the same code cache.

Optimization like: Dead code, escape analysis: Creating Objects in stack, which is never escaped from method, so on.

Form Java 9, Ahead of Time (AOT) compilation is also possible. By JVM Configuration. AOT compiles and creates dot SO files.

If we use AOT, the platform dependence will come into picture.

**Java architecture** Explanation

**Class Loader**

**JVM**

Memory areas allocated by JVM: Class loader, Class area, Heap, Stack, Program Counter Register and Native Method Stack

JVM is platform **dependent**.

**Java Native Interface** (JNI) is often referred to in connection with JVM. JNI is a programming framework that enables Java code running in JVM to communicate with native applications associated with a piece of hardware and specific operating system platform.

**JRE**

Besides the JVM, JRE is composed of a variety of other supporting software tools and features to get the most out of your Java applications.

**JDK** contains all the tools required to compile, debug and run a program developed using the Java platform. JDK includes all the Java tools, executables and binaries needed to run Java programs. This includes JRE, a compiler, a debugger, an archiver and other tools that are used in Java development.

It's a decision by the Java package maintainers to include the Java Virtual Machine (JVM) in the Java Runtime Environment (JRE) which is itself included in the Java Development Kit (JDK).

JVM is included in both JDK and JRE. Refer the JDK\_JRE.png

# Software Architecture

Example: Amazon AWS “Lambda.”

1.Backend as a service (BaaS)

2.Functions as a Service

Application solution that depends on third-party services to manage the complexity of the servers and backend management.

It depends on developing small, independent modular services where each service solves a specific problem or performs a unique task, and these modules communicate with each other through well-defined API to serve the business goal.

communicate via HTTP, hence achieves language independence

Scaling of Monolithic Apps causes all the modules to be scaled instead of on demand modules.

# Java Version

|  |  |
| --- | --- |
| **Version** | **Updates** |
| 1.1 | Inner Classes Java Beans RMI, Remote Methods Innovaction Just in Time (JIT) Compiler |
| 1.2 | J2SE, J2EE, J2ME Collection APIs |
| 1.3 | HotSpot JVM |
| 1.4 | Regular Expression Exception Chaining |
| **1.5** Naming convention Changed **5.0** | AutoBoxing Generics Variable Arguments ForEach Annotations Enumeration |
| 6 Last from Sun | J2SE 6.0 -> Java SE 6 |
| 7 | Strings in Switch statement  Try with resources, auto close resources Multiple Catches, pipe operator |
| 8 | **Define methods in Interface** method reference **Lambda Expression** Final is removed? .As Effectively Final |
| 9 | Ahead-of-Time Compilation Jshell - Java Shell. Jlink -  Removed JavaDB from JDK Made G1 by default |
| 10 | Garbage Collection Interface Local Variable type Inference **Var is introduced** |
| 11 | Epsilon : GC Local Variable syntax for lambda parameter Collection.ToArray(intFunction) Lazy allocation of compiler Threads **Removed Thread.destory(), Thread.stop();** Removed corba |
| 12 |  |
|  |  |
| 14 |  |

# Definition

**Data Structure**

Data structure is a way of organizing and storing data in a computer so that it can be accessed and modified efficiently. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

**Exception**

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

**Object**

Way of defining the state and behaviour for real-world things

**Class**

A class is the blueprint from which individual objects are created.

**Interface**

"An interface provides a means of communication. Like the Mobile phone provides an interface to message and call. It is the responsible of individual brands to define how to interact.

Interface mobile{call(); message(); }

So, if Samsung want to call a gadget a "Mobile" then it should define how it will do the call () and message () behaviours.

Note: all fields are automatically public, static, and final, and all methods that you declare are public.

1. Separate how we use something from how it is implemented.

2. Interfaces are trying to solve a very specific problem by allowing us to interact with objects based on what they do/are, not how they do it.

1.interfaces that completely describe the functionality of a class are usually wrong

2.Interfaces are always implemented by more than one class"

**Abstract class**

"When we want to share code among several closely related classes.

Can have constructors"

**Software Architecture**

Software architecture is the process of converting software characteristics such as flexibility, scalability, feasibility, reusability, and security into a structured solution that meets the technical and the business expectations.

**Inheritance**

Increases the memory footprint: If we inherit a class which has lot of members/variables just to override one method. Then it is overhead.

Not able to inherit more than one class is not the only problem. We cannot borrow anything from the sibling classes.

**Abstraction**

With abstraction, we declare what operations that can be done on an object but how it is done will not be known.

One way of achieving abstraction is by using interfaces.

**Encapsulation**

Member Variables should be private only. These members should be accessed via member functions.

We can make a field as read-only or write-only depending upon the requirements.

We are NOT preventing access to the fields; we are controlling how others can access fields

Private instance and public Methods:

* Do basic Validation.
* Take actions when the field is modified (trigger event).
* Provide thread safety by synchronizing the method.
* Debugging: accessors/properties you can just add a trace inside the function you want or a breakpoint

**Polymorphism**

**Association**

Relationship between two objects is referred as an association.

- an association is known as composition when one object owns other

- an association is known as aggregation when one object uses another object.

Association is denoted by the simple arrow

**Composition**

In UML notation, a composition is denoted by a ***filled*** diamond.

class Company {

Employee emp = new Employee();

}

**Aggregation**

In UML notation, an aggregation is denoted by an ***empty*** diamond.

class Company {

Employee emp ;

Company(Employee e){

emp=e;

}

}

**Multi**-**Tasking**

Ability to execute more than one task at the same time by a single processor. It is often done by some algorithms by OS.RoundRobin.

**Multi-Processing**

It is same as multitasking, however in multiprocessing more than one CPUs are involved.

**Multi-Threading**

It is a way of executing multiple threads simultaneously in a process. Concurrency

**Parallel** **Processing**

Processing of a single program instructions by dividing them among multiple processors with the objective of running a program in less time.

**UML**

Intended to provide a standard way to visualize the design of a system.

**Class Diagram**

It describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

**Object Diagram**

It will show how objects in your system are interacting with each other at some point in time, and what values those objects contain when the program is in this state.

**Sequence Diagram**

It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

**Coupling**

**Cohesion**

**Framework**

Tasks includes creating objects, destroying them, and invoking certain methods of the object at different stages of its lifecycle.

# Concepts

## Instructions Reordering

## Enhanced for Loop

For-each loop is not appropriate when you want to modify the looping array. In case of primitive, but for objects?

User cannot keep track of index if For-each is used.

For-each only iterates forward over the array in single steps. No reverse order. No jumping.

## Pass by value or Reference

Java passes everything by value. With primitives, you get a copy of the contents. With references you get a copy of the contents/Handle/pointer.

<https://javaranch.com/campfire/StoryPassBy.jsp>

## Declared or effectively final

A non-final local variable whose value never changes after initialization is called “Effectively Final”.

\* Variables used inside the lambda should be final or effectivelyFinal.

\*/

**public** **class** EffectivelyFinal {

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

String finalVariable = "final local variable";

// finalVariable = "Can I change non-final variable before anonymous class";

Runnable runnable = () -> {

System.***out***.println("Using final local variable inside Lambda expression");

System.***out***.println("Value of final variable is : " + finalVariable);

// finalVariable = "Can I change non-final variable inside anonymous class";

};

// finalVariable = "Can I change non-final variable after anonymous class";

}

}

The restriction to effectively final variables prohibit access to dynamically-changing local variables, whose capture would likely introduce concurrency problems.

The problem is only for local variables as they are stored in the separate stack. As instance and static variables are stored in Heap this problem will not be seen. If not, the local variable must be Volatile.

# Interface

## Default Methods

When we want to add new methods/functionalityto an existing Interface without having to modify the already implemented concrete class, we use the ***default*** methods. These default methods should be defined with a default definition.

The use of ***default*** keyword in an Interface is not about the access specifier, it conveys that the method is a default implementation.

When we extend an interface that contains a default method, we can override or not override the default method.

Static method in interface is visible to interface methods only hence these static methods can’t be overridden(like static methods of class)

The original motivation to introduce default methods to Java 8 was the desire to extend the Collections Framework interfaces with lambda-oriented methods without breaking any existing implementations

## Functional Interface

An interface having only single abstract method is called as functional interface. Functional interface can have other methods, but they should be default(method with definition) or static methods.

They can have more abstract methods, but it will break the functional interface rule.

Runnable runnable = () -> {System.out.println("Implementation of a FI, without any arguments");};

runnable.run();

Consumer, Supplier, Function, Predicate and BiFunction are extensively used functional interfaces.

# Lambda expression

Compiler does Type inference to find why type of lambda/data it is.

No new Type is created, like, "LambdaType" interface or class. Because of which backward compatibility is achieved. Otherwise, all those functional Interface must be modified to accept the "LambdaType".

the keyword "this" is a reference to an enclosing instance/Scope

Braces and return statements are optional in one-line lambda bodies.

Any local variable, formal parameter, or exception parameter used but not declared in a lambda expression must either be declared final or be effectively final.

Practice the use of functional interfaces with lambda.

# Streams

Stream package contains classes for processing sequences of elements.

Streams can be created using the existing data-provider sources(Collection or array) and can be operated upon or closed only once.

If data-provider source has zero elements, the stream will be empty.

They are divided into 3 parts :

* Source
* zero or more intermediate operation
* terminal operations

**Stream vs Iterator** :

Streams allows us to concentrate on operation's logic, but not on the iteration over the sequence of elements. With Iterator, user must explicitly handle the item while traversing by using hasNext() and next().

Stream package provide built in methods for many operations like:

**Stream** :

* parallelStream() : parallel processing.
* map(): For each element, Function is applied and returns a stream.

**Collectors**:

* partitioningBy : takes a Perdicate and return a map of (Boolean & List )
  + Ex: Collectors.partitioningBy (num -> num > 3)
  + Output : {false=[1, 2, 3], true=[4, 5, 6, 7, 8, 9, 10]}
* groupingBy : takes a Function and return a map of (Type & List )

**Optional**:

orElse: we can set some default value

orElseGet : returning empty stream

# Memory

Stack memory only contains local primitive variables and reference variables to objects in heap space.

Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution

When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error

Refer the ComputerMemory.png in \cognitiveLearning\Resources\Image\garbageCollection.

Size of Object Reference: It is not a part of JLS or JVM Spec, since it is an address: 32 bit on 32 bit CPU, 64 bit on a 64 bit CPU.

# Strings

Strings are immutable. Because of which :

* They are used in cases where immutability is important. Like, hashtable.
* String pool is safe as no one can change the object of string once it gets created.

# Regular Expression

|  |  |
| --- | --- |
| **Expression** | **Description** |
| [abc] | Find one character from the options between the brackets |
| [^abc] | Find one character NOT between the brackets |
| [abc] | Find one character from the options between the brackets |
| ^ | Matches the beginning of the line. |
| $ | Matches the end of the line. |
| . | Matches any single character except newline. Using **m** option allows it to match the newline as well. |
| [...] | Matches any single character in brackets. |
| [^...] | Matches any single character not in brackets. |
| \A | Beginning of the entire string. |
| \z | End of the entire string. |
| \Z | End of the entire string except allowable final line terminator. |
| re\* | Matches 0 or more occurrences of the preceding expression. |
| re+ | Matches 1 or more of the previous thing. |
| re? | Matches 0 or 1 occurrence of the preceding expression. |
| re{ n} | Matches exactly n number of occurrences of the preceding expression. |
| re{ n,} | Matches n or more occurrences of the preceding expression. |
| re{ n, m} | Matches at least n and at most m occurrences of the preceding expression. |
| a| b | Matches either a or b. |
| (re) | Groups regular expressions and remembers the matched text. |
| (?: re) | Groups regular expressions without remembering the matched text. |
| (?> re) | Matches the independent pattern without backtracking. |
| \w | Matches the word characters. |
| \W | Matches the nonword characters. |
| \s | Matches the whitespace. Equivalent to [\t\n\r\f]. |
| \S | Matches the nonwhitespace. |
| \d | Matches the digits. Equivalent to [0-9]. |
| \D | Matches the nondigits. |
| \A | Matches the beginning of the string. |
| \Z | Matches the end of the string. If a newline exists, it matches just before newline. |
| \z | Matches the end of the string. |
| \G | Matches the point where the last match finished. |
| \n | Back-reference to capture group number "n". |
| \b | Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets. |
| \B | Matches the nonword boundaries. |
| \n, \t, etc. | Matches newlines, carriage returns, tabs, etc. |
| \Q | Escape (quote) all characters up to \E. |
| \E | Ends quoting begun with \Q. |

Summary of regular-expression constructs:

https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

# Cloning / Copying

A copy constructor in a Java class is a [constructor](https://www.baeldung.com/java-constructors) that**creates an object using another object of the same Java class**.

Cloning is a process of creating an exact copy of an existing object in the memory. ~~using the clone() method of java.lang.Object class.~~ These Objects must implement the Cloneable Interface. If this Object implementing the Cloneable interface, it will create an exact copy of all its primitive and immutable member variable. If original object has any references to other objects as fields, then only references of those objects are copied into clone object.

**Shallow copy**

The default behaviour of the clone() method is called shallow copy. That is creating an exact copy of all the primitive and immutable member variable of an Object.

**Deep Copy**

In additional to the Shallow copy, if original object has any references to **other** objects as fields, then copy of those objects are also created by calling clone() method on them. So those **other** objects should also implement Cloneable Interface

**Lazy copy**

A lazy copy can be defined as a combination of both shallow copy and deep copy. The mechanism follows a simple approach – at the initial state, shallow copy approach is used. A **counter is also used to keep a track on how many objects share the data**. When the program wants to modify the original object, it checks whether the object is shared or not. If the object is shared, then the deep copy mechanism is initiated.

# Immutable

Safe to use in cache.

Thread Safe, state of object(not the reference) will remain same as nobody can change it.

When to use immutable classes: Notifier events, to avoid method from changing the state, Cache, HashMap.

Immutable classes promote object proliferation, but mutable classes create many defensive copies too.

Guidelines to create immutable class:

* Class and instance variable should be final.
* Constructors should perform deep copy.
* No setters, Getters should return a deep copy of instance variable.

Primitives, String, numeric wrapper objects etc are immutable

Refer the class concepts.immutable.thread.problems.java

# Exception

Throwable is the parent class of Java Exceptions Hierarchy, and it has two child objects – Error(unchecked) and Exception.

Exceptions are further divided into checked exceptions and Unchecked (RuntimeException).

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

**Checked**: Exceptional conditions that a well-written application should anticipate and recover from.

**Unchecked**: Where the application usually cannot anticipate or recover from.

Error: These are exceptional conditions that are External to the application. Unable to read the file because of a hardware or system malfunction

Runtime Exception: Usually indicate programming bugs, such as logic errors or improper use of an API

If the current thread is interrupted/killed <or> if the JVM exits while executing the try or catch then the finally block \*may\* not executed.

**Unchecked exceptions**:

**Errors**: OutOfMemoryError, StackOverflowError

**Runtime** Exception: ArrayIndexOutOfBoundException

**Checked Exceptions**:

**FileNotFoundException**

### Enhanced try block

**try/finally vs tryWithResources**

The method throws finally block exception and suppress the try block exception.

the method throws try block exception and suppress the tryWithResources exception.

The suppressed exception can be retrieved by Throwable.getSuppressed()

\*\*\*\*\*

catch (Exception 1 | Exception2 exp) the catch parameter exp is final and therefore you cannot assign any values to it within the catch block.

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

Wrapping exception. Good Idea.

try { // do something } catch (NumberFormatException exp) {

throw new MyBusinessException(""A message that describes the error."", e); }

# Reflection

The name reflection is used to **describe code** which can inspect other code in the same system (or itself) and to make modifications at runtime.

The ability to inspect the code in the system and see object types is not reflection, but rather Type Introspection.

Reflection is then the ability to make modifications at runtime by making use of introspection.

The distinction is necessary here as some languages support introspection, but do not support reflection. One such example is C++

Things we can find using reflection : https://www.baeldung.com/java-reflection

# Enumeration

You should use Enum types any time you need to represent a **fixed** set of constants that we know by compile time.

Used to represent a group of named constants.

Because enums extends java.lang.Enum so it cannot extend anything else.

# Serialization

To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted into a copy of the object.

**Allowed changes to class after serialization:**

Adding new variables to the class

Changing the variables from transient to non-transient or static to non-static.

Static, non-static anonymous inner class serialization?

behaviour of transient to non-transient or static to non-static?

**Custom overrides**:

readObject(ObjectInputStream ois):ObjectInputStream readObject() method will use this method for reading the object from stream.

writeObject(ObjectOutputStream oos):ObjectOutputStream writeObject() method will use this method for writing the object to stream. One of the common usages is to obscure the object variables to maintain data integrity.

Object writeReplace():After serialization process this method is called and the object returned is serialized to the stream.

Object readResolve():After deserialization process, this method is called to return the final object to the caller program. One of the usages of this method is to implement Singleton pattern with Serialized classes.

# Reference

These subclasses of Reference interact with the garbage collector in different ways.

### Strong

This reference is a normal reference that protects the referred object from collection by GC. i.e. Never garbage collects.

### Soft

These objects are GCed when they can only be reached by the SoftReference **and** there is a shortage of memory.

Typical use case example is keeping a parsed form of a contents from a file. You'd implement a system where you'd load a file, parse it, and keep a SoftReference to the root object of the parsed representation. Next time you need the file, you'll try to retrieve it through the SoftReference. If you can retrieve it, you spared yourself another load/parse, and if the GC cleared it in the meantime, you reload it. That way, you utilize free memory for performance optimization, but don't risk an OOME

MyClass obj = new MyClass ();

// creating a weak reference of type MyClass

SoftReference<MyClass> sobj = new SoftReference<>(obj);

System.out.println ("-> Calling Display Function using strong object:");

obj.Display ();

System.out.println ("-> Object set to null");

obj = null;

// Calling the get() method

obj = sobj.get();//returns null if GC has removed **even** the soft refernce

System.out.println ("-> Calling Display Function after retrieving from soft Object");

obj.Display ();

In the above example, Garbage collector did not reclaim the memory because it has not yet in shortage of memory. Hence, we can retrieve the nullified object and call its methods.

Reference queues are designed for making us aware of actions performed by the Garbage Collector. It appends a reference object to a reference queue as it decides to remove the referent of this reference.

ReferenceQueue<StringBuilder> referenceQueue = new ReferenceQueue<>();

SoftReference<StringBuilder> reference2 = new SoftReference<>(builder, referenceQueue);

### Weak

The object which can only be reached by the WeakReference. Such objects will be GCed and Object reference is appended in the referenceQueue

Sometimes the difference between weak and soft references is unclear.

### Phantom

Adsadasd

**Difference**:

The Strong reference will never be GCed. Only after nullifying it will be eligible for GC. It can never be retrieved even if GC has not reclaimed the memory.

Weak reference Objects will be eligible for GC if it is only reachable by the WeakReference. Once program nullifies the referent, it will be eligible for GC. If it is not yet GCed it can be retrieved.

Soft reference Objects will be eligible for GC if it is only reachable by the SoftReference. Once program nullifies the referent, it will be eligible for GC. If it is not yet GCed it can be retrieved. SoftReference object will only be GCed if there is shortage of memory.

A reference is Phantom reference if it is neither strongly, softly, nor weakly reachable. Once program nullifies the referent, it will be eligible for GC. And can never be retrieved.

# Garbage Collection

How is garbage formed:

* By nullifying a reference.
* Assigning a new object to a reference. Immutable Objects create more garbage
* Anonymous class, Lambda
* Objects created in methods. Even though the reference is in stack, actual object is in heap.
* Cyclic dependencies are not counted as the reference so if object A has a reference to object B and object B has a reference to Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.
* If an object has only weak references it will be eligible for garbage collection.

Basic steps of GC :

* Mark
* Sweep/delete
* Compacting(Is this optional ?)

Garbage collection is a process of looking at the Heap memory, identifying (also known as “marking”) the unreachable objects, and destroying (Sweep) them and followed by compaction of the Heap. An issue with this approach is that, as the number of objects increases, the Garbage Collection time increasing, as it needs to go through the entire list of objects, looking for the unreachable object. So, we need strategic approach about balancing the unreachable objects in the heap and time taken to perform GC.

Depending on finalize method(which will be called when any object is about to be GCed)to do the clean-up of used resources is not good as we are unaware when GC would be performed.

In most configurations the OS allocates the heap in advance to be managed by the JVM hence Global synchronization with the operating system is not needed for every single object creation or deletion. The heap grows or shrinks (using the available virtual space) to a size that supports the chosen throughput goal by changing the maximum pause time.

Since most of the objects are short-lived the Heap space is divided into generations like Young Generation, Old or Tenured Generation, and Permanent Generation. Young Generation is further divided into three parts known as Eden space, Survivor 1 and Survivor 2 space.

Minor GC refers to the GC that happens in the Young Generation and Major GC refers to the GC that happens in entire heap(Young and Old).

Every new Object is created in the Eden space of Young Generation and are moved to surival01 or survial02 after the minor GC if they are still reachable. A counter is maintained for each object indicating the number of Minor GC it has survived. Once a threshold(MaxTenureThresold, usually 16) is reached they are moved to Old Generation. Permanent generation is used to store Metadata related to classes and methods in JVM, it also hosts a String pool provided by JVM

When Old Generation is about to be full, the Major GC will be triggered. When such Major GC happens, the application may pause for few seconds.

OutOfMemoryError is thrown after a Major GC ~~and If more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered.~~

Why the Explicit call to System.gc() is not always guaranteed , may be because GC are configured to run when

* When certain percentage of heap is occupied.
* and

**Performance of GC**: Latency and ThroughPut

**Compaction**

Java avoids memory fragmentation by executing compaction (~ hard-disk defragmentation) at the end of a successful GC cycle. Downside-->longer GC cycle.

Reducing the Impact of Compacting: Compacting is applied only after certain percentage of fragmentation is seen or compacting is stopped when certain percentage of continuous memory is available.

**Throughput** is the percentage of time the application code is running than the time spent in garbage collection, considered over long periods of time. Throughput includes time spent in allocation.

**Latency** is the responsiveness of an application. The pause due to garbage collection affects the responsiveness of applications.

**Footprint** is the size of each generation, which effects the throughput and Latency.

## Types of Garbage Collectors

Four different garbage collectors.

### Serial GC

This GC runs on a single thread and performs the basic steps of GC in a stop-the-world approach.

The first step of this algorithm is to mark the surviving objects in the old generation. Then, it checks the heap from the front and leaves only the surviving ones behind (sweep). In the last step, it fills up the heap from the front with the objects so that the objects are piled up consecutively(compacting) and divides the heap into two parts: one with objects and one without objects (compact).

### Parallel GC

Almost same as Serial collector but uses **multiple threads/cores** so it will be faster. Starts only when heap is almost full. Stops the world and uses multiple threads and CPUs to perform mark, sweep and compacting.

### Concurrent Mark & Sweep (CMS) GC

**Understand: mark vs Remark. When it is concurrent and when it is STW. Does it Compact.**

Stops the world only during Mark . i.e the sweep and compacting operations runs along with the application.

It runs in concurrently with the application threads, but It uses Stop-The-World (STW) approach in two cases.

1.the **GC root** objects are marked as alive. During this phase, all threads of the application are suspended.

2.During concurrent marking, the marked root objects are traversed, and all reachable objects are marked.

3.In the final marking, all threads are suspended and all remaining newly allocated objects are marked as alive.

### Garbage First GC

G1 collector partitions the heap into a set of equal-sized heap regions(instead of Young & Old), each a contiguous range of virtual memory. When performing garbage collections, G1 shows a concurrent global marking phase to determine the liveness of objects throughout the heap. After the mark phase is completed, G1 knows which regions are mostly empty. It collects in these areas first, which usually yields a significant amount of free space.

Ex: Oracle’s HotSpot is well known . It has all 4 types of GCs . they all are used as configured.

CMS is used when less Latency is important, and more memory & CPU is available. Parallel is used if latency is acceptable(websites) and memory and CPUs are less

CMS : GUI

PGC : backend and dataBase

Using JVM config/flags we can specify which type of GC to be used for each generation.

The choice of when to use compaction, how many threads are used in major GC and Minor GC, which type of GC to be used for each generation and some other configs differ in each JVMs and their flags.

Other latest types of GCs are Epsilon, Shenandoah and Z Garbage Collector

# Compare Objects

## Comparable

A comparable object can compare itself with another object of same type. These objects itself must know how it is to be ordered.

So, the class must implement the java.lang.Comparable interface to compare its instances.

Hence, we can only have one logic of comparing objects. To have more than one way of comparing objects we must use Comparator.

public int compareTo(Object obj2){

return s1.compareTo(s1);

}

## Comparator

Unlike Comparable, Comparator is external to the element type we are comparing. It’s a separate class. We create multiple separate classes (that implement Comparator) to compare by different members. We can even compare objects of different Type.

public int compare(Object obj1, Object obj2){

return obj1.compareTo(obj2);

}

Comparator class provides useful methods.

**comparing**() , which returns a comparator Object if the instance variable has a getter methods. Ex: Comparator.comparing(Employee::getName()), returns an Employee type comparator which compares based on the *name* instance variable.

# Data Structures

## Binary Tree

Strict Binary Tree:

each node has 2 0r zero nodes

**Complete Binary tree**:

All nodes are filled and all nodes which are not filled should be as left as possible

Left Child: (i\*2) +1

Right Child =(i\*2) +2

**Perfect Binary tree**:

All nodes have 2 children. other node is a leaf

Root node is at level 0

Max nodes at level i=2^i

## Linked List

Unfortunately, linked lists do not perform very well. Each element in the list is a separate object, and these objects can be spread out all over the computer's memory. CPUs are much faster at accessing data sequentially, so you will get a lot higher performance out of a list implemented on top of an array. An array stores data sequentially. The CPU caches can load bigger chunks of the array into the cache at a time, and have the CPU access the data directly in the CPU cache once loaded.

# Sorting

## HeapSort

Parent : (i-1)/2

Left Child : (i\*2) +1

Right Child =( i\*2) +2

Max number of nodes in Complete binary tree: 2 power h+1, where h is the height of node.

If there are n nodes in complete binary tree or binary tree, then the height of the tree is log.n

\*In complete Binary tree all the leaves will be at (n/2) +1 to n

\*All leaves are considered as hepified.

Because of the above two points we hepify the elements for 0 to (n/2)-1

Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called heap

Max heap always finds the position for largest element

# Collections

<https://www.baeldung.com/java-collections>

Recap/Learn using the above link

## Concepts

### HashCode & Equals

Not every time we need Comparator or Comparable interfaces to differentiate objects of same class. In HashMap and Hashtable,we can just override the hashcode () and equals () to achieve it.

## Iterable

An interface which tells that the collection is Iterable. And to Iterate that collection we can get the Iterator using the methods Iterator ().

The forEach() provides each way to iterate.

It provides a common interface.

Except Map interface all other collection implements Iterable.

## Iterator

It has 3 methods.hasNext(), next(), remove().

By default, remove() throws UnsupportedOperationException. As it is not a good idea for an iterator to perform any operation other than reading.

We can have multiple Iterator for the same Aggregate Object to have different kind of traversing.

**Fast fail**: while iterating through the collection any structural modification by others causes the iterator to throw ConcurrentModificationException.

**Fail Safe**: Iterator makes copy of the internal data structure (object array) and iterates over the copied (may get stale in multi-thread environment) data structure. Ex: CopyOnWriteArrayList.

When we are using any of the modify methods – such as add() or remove() – the whole content of the CopyOnWriteArrayList is copied into the new internal copy.

### ListIterator

ListIterator provides more navigational methods like previous(), hasPrevious() and previousIndex()

## Collection

Collections.synchronizedList(list)

### List

#### Vector

\*Thread Safe, \*Internally uses Array.

Most of the methods are synchronized, causing delay and not atomic level sync

the capacityIncrement(2nd argument) is less than or equal to zero, the capacity of the vector is doubled each time it needs to grow.

#### ArrayList

\*Internally uses Array.

\*Best when read operation are more because of index based.

\*Not synchronized but can get a sync list with the help of collection util. This sync list is slow as other threads must wait while one is writing.

\*Creates an arraylist of size 10 by default otherwise of the specified size.

\*If the size is full while adding, the ensureCapacity() increases the size by half and copies the arraylist in the new ArrayList(Using Arrays.copyOf())

\*trimToSize(). Shrink the capacity of this ArrayList instance to be the list’s current size.

If the ArrayList is created without defining the size and when the size of the arraylist must be increased, the new size is defined as

n = n + n/2 + 1 (verify this)

add & remove methods cause a complete rearrangement of the ArrayList. Complexity: O(n)

#### LinkedList

\*Used when frequent operation is adding or removing elements in the middle of the List

\* Implements Deque, List

### Queue

The parallel methods of offer, peek and poll are add, element, remove. But they are different return values.

The inherited methods from Collection interface throw exception when

1. Add: exception if add task cannot be performed.
2. Element : Exception when queue is empty
3. Remove : exception if queue is empty.

#### Priority Queue

The Priority queue does not take null as input because it must compare the data with the existing queue in order to maintain the order(Natural or Comparator).

The head of the queue is the least element based on the ordering.

#### DeQue

Besides methods like offer and peek which are available from Queue class, there are other methods which helps to remove/add/access the elements from both the ends of the queue.

* peekFirst, peeklast
* offerFirst , offerLast
* pollFirst, pollLast

### Map

HashMap does not maintain any order.

LinkedHashMap maintains the insertion order.

TreeMap sort the entries in ascending order of keys.

linked hash map reduces the chaos in the ordering of a hash map without incurring the performance penalty of a tree map and also provides insertion order.

**compute method:**

The compute(Key, BiFunction) method of Map allows you to update a value. The compute() method tries to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping is found). This method is used to automatically update a value for given key in Map.

map.put("Address", "Kolkata"); System.out.println("Map: " + map);

map.**compute**("Address", (key, val)-> val.concat(" West-Bengal"));

System.out.println("New Map: " + map);

Output

Map: {Address=Kolkata} New Map: {Address=Kolkata West-Bengal}

**Merge method** :

If the mapping for a given key(First param) doesn't exist, or its value is null, it associates the key with the provided value(second parameter).

Otherwise, it calculates a new value using the remapping function and updates the mapping accordingly. In the below case it is calling sum() in Long class.

Map<T, Long> addToValue = new HashMap<>();

inputList.forEach(e -> **resultMap.merge(e, 1L, Long::sum)**);

Note:

The order of inputs to the remapping function.

1. value of key in the resultant map

2. value of key being searched in the source map

Ex: puzzles.collection.map.MergeMaps

Linked hash map reduces the chaos in the ordering of a hash map without incurring the performance penalty of a tree map.

**Iterating on Maps**: http://www.sergiy.ca/how-to-iterate-over-a-map-in-java

Map is not Iterable, and it is not a collection. But we get Set from a Map and then Iterate over it.

Not synchronized but can get a sync list with the help of collection util. This sync list is slow as other threads must wait while one is writing.

But Sync list is prefered(May be ) over concurrentHashMap if there are less reads and more write.

ConcurrentHashMap does not allow null keys or values. So, they are NOT equal alternatives of a synchronized map.

#### HashMap

Internally implemented using arrays and linkedList. The linked list is dynamically replaced with BST once the map reaches a threshold (after the number of collisions in each bucket location exceed a certain threshold(8)).

**What is collision :** A collision occurs when two keys are hashed to the same index

Arguments(capacity,loadFactor)

Capacity: intial capacity. Which can be greater or equal to map.size().

Ex: collection.arrayList.Capacity

Load factor : how full the hash table is allowed to get before its capacity is automatically increased

Default: load factor is 0.75, initial capacity 16.

Resizing means capacity will be doubled(2^n) and Rehashing.

put() returns the old value if we add the same key again. And overrides the old values.

When put() returns null, it could also mean that the previous value associated with the key is null, not necessarily that it’s a new key-value.

Same with get(), if get() return null, it may be possible that the value is null. So, use containsKey() in such cases.

\*hash() return zero for a null key, so stores one null key, at the first location.

\*entrySet() and keyset() methods returns the collection view, which is backed by HashMap even though Maps are not Collection(does not implement Collection)

HashMap has an inner class called an Entry Class which holds the key and values.

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

{

final K key;

V value;

Entry<K,V> next;

final int hash;

}

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

**put()**

* hash(key.hashCode())
* indexFor() method is used to get the exact location(bucket) to store the Entry object.
* Objects(Entry Objects) in the bucket are linked together using the instance variable(Entry<K,V> next) of the Entry class. Which is used during collision, to traverse.

#### LinkedHashMap

* OverComes the drawback of HashMap by maintaining the insertion order using two added pointers.
* It maintains a 3rd argument , if set to true, the least accessed element is listed first.

class Entry<K,V> **extends** HashMap.Node<K,V> {

Entry<K,V> before, after;

Entry(int hash, K key, V value, Node<K,V> next) {

super(hash, key, value, next);

}

}

LinkedHasHashMap extends HashMap function (insertion order)

#### Hashtable

Hashtable doesn’t allow any null key or value.

It is synchronized hence thread safe and slow. In many ways it is similar to HashMap.

Instead of hashtable use concurrentMap.

Hashtable ht = new Hashtable ();

ht.put(null, null);

#### TreeMap

Learn something 😉

### Set

Set returns true to indicate that the object is added into the map.

return map.put(e, PRESENT)==null; //PRESENT, Dummy value to associate with an Object in the backing Map

#### TreeSet

1. Order: Ascending
2. Null : not allowed

#### HashSet

1. Order: Ascending
2. Null : not allowed

Factors of performance in Hashset : Initial Capacity and Load Factor.

The load factor describes what is the maximum fill level, above which, a set will need to be resized and rehashed.

We can also create a HashSet with custom values for initial capacity and load factor:

Set<String> hashset = new HashSet<>();

Set<String> hashset = new HashSet<>(20);

Set<String> hashset = new HashSet<>(20, 0.5f);

In the first case, the default values are used – the initial capacity of 16 and the load factor of 0.75.

A low initial capacity reduces space complexity but increases the frequency of the expensive process of rehashing. On the other hand, a high initial capacity increases the cost of iteration and the initial memory consumption. So the input data and its iteration frequency has to be considered.

The expected time complexity of adding an element to a set is O(1) which can drop to O(n) in the worst case scenario (only one bucket present) – therefore, it's essential to maintain the right HashSet's capacity.

# Complexity

## Standard Algorithms

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Space Complexity** |
|  | **Worst** | **Worst** |
| Mergesort | n log(n) | n |
| Timsort | n log(n) | n |
| Heapsort | n log(n) | 1 |
|  |  |  |
| Quicksort | n^2 or nlogn(randomize) | log n |
| Bubble Sort | n^2 | 1 |
| Insertion Sort | n^2 | 1 |
| Selection Sort | n^2 | 1 |
|  |  |  |
| Tree Sort | n^2 | n |
| Bucket Sort | n^2 | n |
| Shell Sort | n(log(n))^2) | 1 |
| Radix Sort | nk | n+k |
| Counting Sort | n+k | k |
| Cubesort | n log(n) | n |
|  |  |  |

## Order of Complexity

|  |  |
| --- | --- |
|  | |
| one | Constant time means the running time is constant |
| log n | Logarithmic : algorithm divides the problem into sub problems with the same size(half).  In programming context, the base of log is 2 (not 10), so O(log n) scales like 1 sec for 10 elements, 2 sec for 20, 3 for 40 etc |
| Sqrt n |  |
| n | Linear - When an algorithm accepts n input size, it would perform n operations as well. |
| n log n | Linearithmic - which divide the problem into sub problems recursively and then merge them in n time |
| n^2 |  |
| n^3 |  |
| 2^n |  |
| n! |  |

In logarithm graph time curve decelerates as n increases.

https://stackoverflow.com/questions/2307283/what-does-olog-n-mean-exactlyIn

Logarithm is essentially the inverse of exponentiation.

Now, if you can prove, that at every iteration of your algorithm you cut off a fraction of this space,

that is no less than some limit, this means that your algorithm is running in O(logN) time.

O(log n) running times are very common in any sort of divide-and-conquer application, because you are (ideally) cutting the work in half every time.

Then why not log(N/2)

# Design Pattern

Types of DP: 3, Behavioural, Structural, Creational

Number of DP: 11 + 7 + 5

1.Design Pattern is a template that must be implemented to handle a problem

2.Some pattern's may be very similar. When confused, first focus on the Type/intent of the pattern.

Understand the design principle properly with would lead to these kinds of solutions.

Maintainable & Expandable

Clean & Readable

**During Interview**

When asked about a designing an application, they are checking:

1.Decomposing larger problems into smaller ones.

2.Creating a structured hierarchy, or graph, or parts, defining components.

3.Analyzing functional requirements per component.

4.Mapping components and inter-component relationship to objects and services.

## Behavioural

Behavioural patterns are used in communications between entities and make it easier and more flexible for these entities to communicate.

uses abstract classes or interface with composition to implement.

### Chain of Responsibility

**GenericActionController:**

1. Chain of Responsibility
2. Factory pattern
3. Observer Pattern
4. Facade Pattern

### Strategy

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Used when there are several types of algorithms that can be used to perform particular a task. Ex:Sorting.

https://dzone.com/articles/design-patterns-the-strategy-and-factory-patterns

A factory pattern is used to create objects of a specific type. A strategy pattern is used to perform an operation (or set of operations) in a particular manner.

Bike bike = new factoryPattern(“Pulsar”);

bike.startStrategy(“KickStart”);

bike.startStrategy(“SelfStart”);

This bike example uses a Static/Simple Factory. To understand the difference between strategy and factory, see what Factory pattern is.

Unclear about the need of following separation

**Runtime Strategy Selection**

1.Conditional Logic: using some token (i.e., a supplied string, integer, etc.) and generating the concrete strategy object that corresponds to the supplied token.

2.Reflection

**Static Strategy Selection**

1.Reflection

2.Dependency Injection

### Observer

*The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*

Make sure to deRegsister. Memory loss.

Slight Modification to the origin pattern:

If the observer is created by passing the observable (concrete, so that we can access the methods/getters), we can avoid the notify method's arguments. Observer can fetch the details from the observable object passed during creation

### Command Pattern

The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue, or log requests, and support undoable operations.

Command Pattern intends to encapsulate in an object all the data required for performing a given action (command), including what method to call, the method's arguments, and the object to which the method belongs.

The pattern encapsulates everything required to take an action and allows the execution of the action to occur completely independently of any of that context. If that is not a requirement for you then the pattern is probably not helpful for your problem space

They can have parameterised constructor but is it possible to know the parameters at the time of command creation?

**Returning the result**:

The command or the receiver can return the result in 2 ways.

1. Either by observer pattern

2. the invoker object passing a Result object as a argument to the method call so that command or Receiver object loading the result.

Even though it is possible to pass arguments and return value from command, it is not recommended because they are supposed to work independently.

**Members**:

Invoker, command, receiver

**UseCases**:

In some cases, the invoker also stores and queues commands, besides executing them. This is useful for implementing some additional features, undo/redo functionality.

Are the commands supposed to do some preProcessing/PostProcessing before/after invoking the Receiver? If not, why is the invoker calling the command. Cannot it directly call receiver. Yes, pre and post work may be something like dataBase open and close

**undo/Redo**:

Use 2 stacks, undo, and redo stack. If you undo an action, it pops from the undo stack and pushes onto the redo stack. Adding a new action is pushing a new one onto the undo stack and clearing the redo stack

https://stackoverflow.com/questions/1154935/command-pattern-returning-status

...................understand: Command pattern using generic parameters

### Iterator Pattern

*The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.*

Iterator Pattern provides a way to access the elements (sequentially?) of an aggregate object without exposing the underlying structure.

with Iterator pattern, we should be able to Iterator in any aggregation of Objects. Ex: List Employee objects

What is the need of Iterable intertace, why can’t we directly get the iterator?

Imagine Iterable inteface is not there. so to get the iterator of (say) Employee::getEmployeeIterator(), Student::getStudentlterator(), getTeacherlterator(). So Iterable provides a unified API.

Use factory pattern to return one of many different types of Iterators based on some condition.

**Advantages**:

1. Hides internal collection type.

2. The Iterator remembers the current position

3.enhanced For loop

When creating the Iterator,

1. The Iterable passes the itself(this): Company passes class

2. or just the collection.

Use factory pattern to return one of many different types of Iterators based on some condition.

Iterators can be implemented in 2 ways:

1.Works on the original copy of the collection. This could case runtime exception (CurrentModification) when the someone modifies the collection while Iterating.

2. Works on a copy of collection to avoid the above problem. But may have stale data.

3. Or work on original collection and listen to the modification of the collection and update accordingly

javapapers.com/design-patterns/iterator-design-pattern/

**STB example**:

Program banner and complete Event list. At a given point in time the program banner needs to know only one event data. It needs to know if the next and previous event exists. So, iterator is the best.

Code this banner Iterator to understand better.

### Template Method

***The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.***

Members: Abstract class and its concrete class!!

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

The template method is declared **final** to prevent subclasses Changing the sequence of steps in the algorithm. Hence the flow/control on the algorithm is stronger in Template than in Strategy

The UML of strategy pattern is same as interfaces & Concrete Class.

The UML of Template is like Abstract class

Java has the concept of Abstract class but other languages might does not have.

**Example**:

1. Tax computing website, they ask if we have any investment, HRA...And then calculate

2. WebPage template

### State Pattern

***The State Pattern allows an object to alter its behaviour when its internal state changes. The object will appear to change its class.***

The State Pattern allows an object to alter its behaviour when its internal state changes.

When the concrete state object about to set the next state, it is better NOT to set the state (constant or new StateObject() directly . Instead use the help of factory or dependence injection to avoid class coupling

Ex: to set the nextstate : getProcessingState()

Strategy and state have similar UML.

State Machine is different from state design pattern

Code an example of State Pattern:

Game. Make the character to walk, talk, run, fight. Pass command and change the state. Change state A-> B, A->A

## Structural

***These design patterns are all about Class and Object composition. Structural object-patterns define ways to compose objects to obtain new functionality.***

https://stackoverflow.com/questions/350404/how-do-the-proxy-decorator-adapter-and-bridge-patterns-differ

<https://www.youtube.com/watch?v=lPsSL6_7NBg>

### Decorator Pattern

The Decorator Pattern attaches additional responsibilities to an object dynamically at compile time or runtime time using by wrapping(constructor) objects. Decorators provide a flexible alternative than sub classing to extending functionality.

There are 3 entities in Decorator pattern.

1. The basic Interface.

2. One or many concrete Class that provide the basic functionalities.

3. Decorator Class/Classes that takes/wraps the concrete class (as a constructor argument) and provides the addition functionalities.

The Decorator class should implement the basic interface too (Why---It will be easy to Use the basic Interface reference and call the operations)

Decorators should not be inter-dependable.

**DrawBacks**:

1. All methods in the decorated interface must be implemented in the decorator class. Can this drawback be solved by combining the command pattern? i.e. the Decorators should implement command pattern. May be possible in specific case.

Or maybe create an abstract base class for all the Decorator classes.

2. This pattern is only good if there are many decorators

you are trading "have to write pass-throughs for every method, not just the ones you're changing & do 2 step object creation", for "have to write a subclass for each concrete class you want to change".

Very Useful DP to extend the functionality of legacy class without disturbing the class.

The complexity in decorating the objects can be reduced by other means.

Like, using decorator-builder?

This pattern adopts Single Responsibility Principle, Open-Closed Principle, Dependency Inversion.

Ex: FileReader, BufferedReader

https://dzone.com/articles/is-inheritance-dead

<https://dzone.com/articles/the-decorator-builder>

FilteredServiceListCreator and its decorator

Label,with scroller,arrow up & down, pic label

**EPG Guide Colouring**:

Normal, unauthorised, and scrambled service.

Different kinds of decorators: 3 different text styles, 3 different colours, 3 different pic labels.

### Adapter Pattern

***The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.***

The adapter is the solution for classes that do similar jobs but do not have a unified interface. Adapter provides the uniform interface and can be implemented using either multiple inheritance or delegation through embedding a member of the adaptee.

**There are two types of Adapter**

1. Objects Adapters(Composition): Adapter Implements the interface. Adapter holds the object of the other class(Adaptee)

2. Class Adapters (Inheritance): Adapter Implements the interface. Adapter inherits the other class(Adaptee).

You should consider using the Adapter Pattern whenever you want to use an existing class’s functionality, but its interface is not the one that you require.

Interfaces are incompatible, but the inner functionality should be as required.

Decorator and Adapter does wrap already existing object, and such is typically provided in the constructor.

dzone.com/articles/adapter-design-pattern-in-java

Decorator and Adapter wrap existing object, and that is typically provided in the constructor.

### Bridge Pattern

***Decouple an abstraction from its implementation so that the two can vary independently***.

Adapters are used when we encounter a problem, but Bridge is implemented to avoid futuristic problems. A bridge is by design, put in place on purpose. An adaptor is a patch.

Separates the Platform independent from platform dependent. It is the solution whenever there are two orthogonal dimensions in the domain.

allows loose coupling between algorithm and platform

-->what is the difference between decorator and bridge. Why cannot we have multiple decorators

-->Is Bridge pattern is a composite of the Template and Strategy patterns.

-->View/Resource is a factory.

Is Handler/View/Resource ==== a bridge pattern?

Bridge and Adaptor pattern holds a reference of an abstraction. That does mean both patterns are same. To understand the difference:

The key difference between Bridge and Adapter patterns lies in their intents. Adapter focuses on resolving incompatibilities between two existing interfaces. It doesn't focus on how those interfaces are implemented, nor does it consider how they might evolve independently. It's a way of making two independently designed classes work together without reimplementing one or the other. Bridge, on the other hand, bridges an abstraction and its (potentially numerous) implementations. It provides a stable interface to clients even as it lets you vary the classes that implement it. It also accommodates new implementations as the system evolves.

As a result of these differences, Adapter and Bridge are often used at different points in the software lifecycle. An adapter often becomes necessary when you discover that two incompatible classes should work together, generally to avoid replicating code. The coupling is unforeseen. In contrast, the user of a bridge understands up-front that an abstraction must have several implementations, and both may evolve independently. The Adapter pattern makes things work after they're designed; Bridge makes them work before they are.

### Façade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem hide or easier to use.

Ex: **SpeedTestServiceProvider**

* Using the subsystem of downloading.
* Invokes the Generic Action Controller to create a IPEvent,get the catalogueID, AssetInfo,Book the asset, start down.
* Mantains the timer
* Calculate the download speed

### Proxy Pattern

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

Simply speaking, a Proxy object is one through which we control access to the actual object on which the functionality lies.

The access to an object should be controlled.

Additional functionality should be provided when accessing an object.

### Composite

### FlyWeight

## Creational

### Static Factory

Simple factory is otherwise known as static factory.

Only one factory is available.

That one factory is mostly class

### Factory Method

***The Factory Method Pattern defines an interface for creating an object, but let’s subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.***

More than one factory is available. Abstracted by an interface

The Factory Method Pattern defines an interface for creating an object, but let’s subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

The definition says, the subclass of the factory decides which class to instantiate. So, the logic should be in the subclass (The Concrete Class).

A factory pattern is used to create objects of a specific type. A strategy pattern is used to perform an operation (or set of operations) in a particular manner.

**Ex1**: Best is View class and its resource adapters of EPG.

Cafe ,TataSky and Airtel

**Ex2**: Rate a cricket player.

Step1: Get the factory based on player type. i.e, BatsMenFactory or BowlerFactory.

Step2: Once I have that, I use it to get the batting performance calculator.

Step3: This calculator uses different strategy to calculate the batting performance

**When to use**:

--Might need a computation before initialization of object (Like in game, creating different types of enemies at random places and random type of enemy)

--Creating an object often requires complex processes not appropriate to include within a composing object.

--The object's creation may lead to a significant duplication of code,

--may require information not accessible to the composing object,

--may not provide a sufficient level of abstraction,

--may otherwise not be part of the composing object's concerns.

### Abstract Factory

The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory of Factories

The abstract factory pattern provides a way to encapsulate a group of individual factories

Provide an interface for creating families of related or dependent objects

**Always try to be clear on the difference between Factory and AbstractFactory**

Examples of AF:https://stackoverflow.com/questions/2280170/why-do-we-need-abstract-factory-design-pattern

### Singleton

***The Singleton Pattern ensures a class has only one instance and provides a global point of access to it.***

**Eager Initialization vs Lazy Initialization**:

The double check is done because: The lock is grabbed only if the Singleton instance does not exist, and then the existence of the instance is checked again(because, what if another thread has created the instance while this thread is waiting for the lock) in case another thread passed the first check an instant before the current thread. By this, we intend to avoid the expense of grabbing the lock of the Singleton class every time the method is called (Avoiding method Sync). Anyway, this is also not good approach.

In lazy initialization, Singleton is created only when Object is created. But in early initialization, if anything of that class is accessed the singleton object is created i.e., if any other static member or static variable.

In lazy initialization you give a public API to get the instance. In multi-threaded environment it is challenging to avoid unnecessary object creation. So, we put synchronization blocks which poses unnecessary locking to be done to check for object already created. So, it becomes a performance issue in this case. In reality, most use cases this sort of code it will always be executed, so is it worth to handle this overhead of thread issues?

So if we are sure that creating object is not going to take any significant memory and its almost always going to be used in your application then it is good to create in static initialization. Also please do not forget to make your instance final in this case as it make sure that the object creation is reflected properly and in totality to main memory which is important in multi-threaded environment.

public class BillPughSingleton {

private BillPughSingleton(){}

private static class SingletonHelper{

private static final BillPughSingleton INSTANCE = new BillPughSingleton();

}

public static BillPughSingleton getInstance(){

return SingletonHelper.INSTANCE;

}

}

**Need for Singleton**:

1. you do not want to create more than one heavy resource

2. controlling access to a resource. To avoid an inconsistent state, Like in DataBase/audioManager

3. Singletons can be stateful or stateless. Stateful can provide services like maintaining a counter. Stateless can provide utility functions that need no more information than their parameters

Multiple instances is possible because of multiple VM, multiple class Loader

volatile case

**Find these reasons** in --https://www.oracle.com/technical-resources/articles/java/singleton.html

1.If you add database connections or use a JDBC driver that allows multithreading, the Singleton can be easily adjusted to allow more connections.

**Understand the Singleton scenarios when:**

**1.Reflection** ->ENUM.

JVM handles the creation and invocation of enum constructors internally. But

Implementing singletons as enums is a clever technical trick, but it’s misusing the meaning of an enum, isn't it?

The easiest way to think about this is: Are singletons enumerations?

We can inhert, implement when using ENUM ?

class Reflection{

//we can throw exception because ,it is immpossible that the static final instance is null after class loading.

private static final Singleton instance = new Singleton();

private Singleton() {

if (instance != null)

throw new IllegalStateException("Trying to create second instance of this class.No Reflection please");

}

public static Singleton getInstance() {

return instance;

}

}

With Reflection we can break private moidifier of any class so why worry about singleton !!!

**2.Cloning** ->override clone() method and throw an exception from clone method or return the same object.

**3.Serialization** : Implement Readresolve(), this is invoked while deserialzation

protected Object readResolve() {return instance;}

The readResolve method is called when ObjectInputStream has read an object from the stream and is preparing to return it to the caller. ObjectInputStream checks whether the class of the object defines the readResolve method. If the method is defined, the readResolve method is called to allow the object in the stream to designate the object to be returned.

### Builder Pattern

Telescoping constructor pattern: One constructor with only **required** fields and many other constructors with different combinations of optional fields. Difficult to maintain and bug prone.

JavaBeans Pattern: One constructor with only required fields and setters for all other optional patterns. Inconsistent state and supports mutability.

Constructs complex objects using step-by-step approach

The builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters, especially if many of the parameters are optional or of identical type.

Create a basic object with the required fields. Later, add-on the optional fields, as necessary.

We directly call the static inner class constructor and the chain. Finally, we call build () of the static inner class which returns the actual object. Its better to have a private Constructor for the actual class, which will be called by the build () of the builder class.

Perform argument validity check as early as possible, may be when creating the actual object(in build method) from the builder object and throw IllegalArgumentException if any state issues. which is also needed in regular constructors.

Building the object with builder pattern is better than using setters of the class. If setters are used, we cannot make sure that the all the required variables are set. Since build method is mandatory, we are sure that checks are done.

Amazon.com: filters could be implement by Builder pattern

Should the builder be a static inner class?

better, Inner Class because Builder needs to have access to the private properties

In my opinion try avoiding this pattern, look for all possible option to reduce the arguments.

Like, combine arguments and separate it. Create a separate ContactDetails class instead of storing all details in the Employee class a memberVariables.

Defining the Calling Protocol on the Builder Object

<http://www.codinghelmet.com/articles/advances-in-applying-the-builder-design-pattern>

### Prototype

??

# Design Principles

Links to read:

<http://wiki.c2.com/?PrematureOptimization>

## Single responsibility

must change Look for all the reasons a class. If there is more than one reason to change a class, then it means this class does not follow the single responsibility principle.

Low coupling and high cohesion.

Design Pattern : Command Pattern.

this does not imply that each class should have only one method, but they should all relate directly to the sole responsibility of the class.

## Open-Closed Principle

**You should be able to extend a classes behaviour, without modifying it.**

**Apply abstraction to those parts of the program that the designer feels are going to be subject to change.**

No matter how “closed” a module is, there will always be change against which it is NOT closed. So, the designer must choose the kinds of changes against which to close his design.

This principle is more related to the controller class.

The class exhibits its extendable functionality by providing a defined protocol (**Interface/Composition**) instead of adding switch Case/if-else or inheriting/modifying .

**Example**: Chrome browser. It can take any number of extensions. The chrome app does not need any modification but can do more things by adding extensions.

**Design Pattern**: Factory Method and Abstract Factory.

Inheritance is just one of techniques used to fulfil OCP. Strategy pattern, decorator pattern, ordinary composition, Generics etc can be used.

If any functionality (method) depends on only primary properties, we can declare them in the interface. If a functionality depends on an external entity, always use composition rather than inheritance

## Liskov Substitution Principle

Inheritance

Principle states that any method that takes class X as a parameter must be able to work with any subclasses of X.

**Derived classes must be substitutable for their base classes**

The principle makes sure that every class follows the contract defined by its parent class.

Basically, all the concrete classes of a Interface should have method definition. Empty methods are not meaningful

Lean/tends towards inheritance. Does it? Yes. It is about inheritance

<https://www.youtube.com/watch?v=ObHQHszbIcE>

## Interface Segregation Principle

Make fine grained interfaces that are client specific.

Clients should not be forced to depend upon the interfaces that they do not use.

## Dependency Inversion Principle

(Inversion of Control)

Program to an interface, not to an implementation.

Dependency Inversion (DI) or Inversion of Control (IoC) is **achieved** by Dependency Injection

Real-life applications can have hundreds of dependencies scattered across the codebase whose creation and management would need to be centralized.

Helps a lot in Unit testing. We can inject mocked object.

<https://martinfowler.com/articles/injection.html>

# Threads

**Good Read**:

http://tutorials.jenkov.com/java-util-concurrent/index.html

http://tutorials.jenkov.com/java-concurrency/index.html

<http://tutorials.jenkov.com/java-multithreaded-servers/index.html>

https://www.youtube.com/c/DefogTech/playlists

Java memory model specifies how the JVM works with the computer's memory (RAM).

NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, TERMINATED.

**https://www.baeldung.com/java-concurrency**

## Simple Definitions

A program in execution is often referred as process.

A process may consist of multiple threads.

A process has its own address space. A thread uses the process’s address space and share it with the other threads of that process.

A thread can communicate with other thread (of the same process) directly by using methods like wait(), notify(), notifyAll().

A process can communicate with other process by using inter-process communication(IPC).

**Multithreading**: It is a way of executing multiple threads simultaneously in a process.

**Multitasking**: Ability to execute more than one task at the same time by a single processor.

It is often done by some algorithms by OS. **Concurrency**.

**Multiprocessing**: It is same as multitasking, however in multiprocessing more than one CPUs are involved.

**Parallel Processing**: processing of program instructions by dividing them among multiple processors with the objective of running a program in less time.

* When the main method creates a thread, it means there are more than one thread in the environment. This is called multi-Threading.
* The way a single processor runs both(or N) threads(Round Robin or Time slice or etc) is called multi-Tasking.
* multi-Processing is the ability to run more 1 to N threads using 1-N processor .
* Parallel processing is splitting of one task among multiple processors to finish faster.

**Thread Safe Class** : A class is thread-safe if it behaves correctly when accessed from multiple threads.

**Critical section :** A block of code that accesses a shared resource and can't be executed by more than one thread at the same time.

A race condition is a special condition that may occur inside a critical section.

Threads are instances of class java.lang.Thread, or instances of subclasses of Thread class. These are objects like any other class. In addition to being objects, java threads can also execute code. , like a virtual CPU that can execute your Java code - inside your Java application.

## Cost of creating & starting Threads

* it allocates memory for a thread stack that holds a frame for every thread method invocation
* each frame consists of a local variable array, return value, operand stack and constant pool
* some JVMs that support native methods also allocate a native stack.
* each thread gets a program counter that tells it what the current instruction executed by the processor is.
* the system creates a native thread corresponding to the Java thread.
* descriptors relating to the thread are added to the JVM internal data structures.
* the threads share the heap and method area.

## Daemon Thread

Daemon thread is a low priority thread (in context of JVM) that runs in background to perform tasks such as garbage collection (gc) etc., they do not prevent the JVM from exiting. If JVM finds running daemon thread (upon completion of user threads), it terminates the thread and after that shutdown itself(JVM).

A thread should be set to daemon before starting the thread.

Threads are divided into 2 parts, normal thread, and daemon threads. When the JVM starts up, all threads it creates are daemon threads, except the main thread. When a new thread is created, it inherits the status of the thread that created it, so by default any threads created by the main thread are also normal threads.

## Basic wait & notify

When the thread calls the wait() method, the JVM puts the current/this thread to sleep and releases the object that controls the synchronized block of code. This thread will be sleep state until notiy() or notifyAll() is called.

Wait and notify should always be called from a synchronized code otherwise IllegalMonitorStateException is thrown.

## Thread Safety

A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of:

1. The scheduling or interleaving of the execution of those threads by the runtime environment

2. with no additional synchronization or other coordination on the part of the calling code.

The Reference is not Thread Safe.

Locking data so that it can only be accessed by one thread at a time is just one possible technique for creating thread safety

## Volatile

Volatile variables are always read/written from/to the main memory(not from CPU cache or CPU registers)IMMEDIATELY.

More time consuming because cache memory can’t be used.

Volatile variable may be helpful when doing operation which are atomic.32 bit

Accessing volatile variables also prevent instruction reordering which is a normal performance enhancement technique.

When a volatile instance variable is read/written from/into main memory, other instance variables of that class is also read/written.

And Java makes sure that read/write on another instance variable happens after the read/write of the volatile variable. This is called "**Happens-Before**" Guarantee.

There are still situations where declare a variable as volatile is not enough. A race condition could happen between the time a thread decides to write, and the actual write time. So, Synchronization is needed!

## Atomic Objects

**Compare-and-swap (CAS )**

A typical CAS operation works on three operands:

1. The memory location on which to operate (M)
2. The existing expected value (A) of the variable
3. The new value (B) which needs to be set

The CAS operation updates atomically the value in M to B, but only if the existing value in M matches A, otherwise no action is taken.

public void increment() {

while(true) {

int existingValue = getValue();

int newValue = existingValue + 1;

if(counter.compareAndSet(existingValue, newValue)) {

return;

}

}

}

## Blocking & Non-blocking algorithms

An algorithm is said to be non-blocking if the suspension of one thread cannot lead to the suspension of other threads involved in the algorithm.

Blocking algorithms block the thread until the requested action can be performed. Non-blocking algorithms notify the thread requesting the action that the action cannot be performed.

If the algorithm guarding a concurrent data structure is non-blocking, it is said to be a non-blocking algorithm. The data structure is thus said to be a non-blocking, concurrent data structure.

## Synchronize

Synchronization guarantees that only one thread can enter a given ~~critical~~ section of the code at any given time.

**Thread synchronization can be achieved by**:

* Synchronized block
* Locks
* Atomic variables like AtomicInteger
* And many other 😉

The synchronized keyword can be used to mark four different types of blocks:

* Instance methods
* Static methods: synchronized on the Class object of the class
* Code blocks inside instance methods
* Code blocks inside static methods:synchronized(MyClass.class)

Synchronized keyword places ***some*** restrictions on reordering of instructions before, inside and after synchronized blocks.

Do not synchronize on String objects, or any primitive type of wrapper objects. These might be cached or reused internally by the Java compiler, Java VM or Java libraries.

May be, it is not a good idea to sync on immutable objects too. Because we might lose the original object's reference after it is used to sync. In such cases, can a final reference variable help ?

**Synchronized blocks:**

With synchronized block, we can have control on which object lock has to be acquired. Whereas in synchronized methods , the lock is acquired on "this" object.

Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again, regardless of whether the variable is declared volatile or not.

If we have more than one critical section, check if it is possible to locking using different locks.

**Synchronized Block drawBacks**:

Does not allow any other thread, not even for safe reading. Alternative: Read/Write locks

No ordering of threads waiting in the queue.

Slight delay

These block synchronization holds good only in one instance of JVM.

Concurrency utility classes are advanced than synchronized key word

ThreadLocal a hack to avoid sync issues.?

To handle UncaughtExceptionHandler in threads

thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler()

{

public void uncaughtException(Thread thread, Throwable e)

{

System.out.println("Exception caught: " + e);

}

});

**Ways to avoid DeadLock**:

Lock order: If you make sure that all locks are always taken in the same order by any thread

Lock Timeout.

**Causes of STARVATION**:

high priority, indefinitely waiting to enter synchronized block, indefinitely waiting for the notify()

Isn’t thread waiting for lock synchronized block FIFO?

public class **Lock**{

boolean isLocked = false;

Thread lockedBy = null;

int lockedCount = 0;

public synchronized void lock() throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(isLocked && lockedBy != callingThread){

wait();

}

isLocked = true;

lockedCount++;

lockedBy = callingThread;

}

public synchronized void unlock(){

if(Thread.curentThread() == this.lockedBy){

lockedCount--;

if(lockedCount == 0){

isLocked = false;

notify();

}

}

}

}

With a simple Lock, while isLocked is true, the thread calling lock() is parked waiting in the wait() call. In case the thread should return unexpectedly from the wait() call without having received a notify() call (AKA a Spurious Wakeup) the thread re-checks the isLocked condition to see if it is safe to proceed or not, rather than just assume that being awakened means it is safe to proceed.

Google about Lock Class

## Lock Interface

With Lock interface, the restriction of acquiring and releasing of the lock in the same block of code(like when using synchronized keyword) does not apply. This makes it a little flexible to use and more error prone because the responsibility of acquiring as well as releasing the lock is on the programmer.

The semantices of Lock class is like the wait/notify method of Object class.

Always put the critical session in try block and release the lock in the finally block.

### ReEntrantLock

These lock class allows a thread to enter another critical session of code whose access is on the same lock Object. The method getHoldCount() gives the number of holds on this lock by a thread.

The constructor of ReEntrantLock class takes a boolean to provide a fairness in the wait time.

The newCondition() returns a new Condition object. Which can be used for the communication between thread by using the wait & signal methods. Example of Condition object's combinations is the Producer-Consumer usecase.

## Spin Lock

Spin locks does not release the CPU. So, Spin locks are good when we know that the critical section is of very short time.

We don’t have a Java class but JVM configs to support or config these.

## ReadWriteLock Interface

We need to acquire a readlock() and writelock() separately from the ReadWriteLock classs. The lock will be given if :

**Read Access**: If no threads are writing, and no threads have requested write access.

**Write Access**: If no threads are reading or writing.

writeAccesses, writeRequests, writingThread, Map<Thread, Integer> readingThreads = new HashMap<Thread, Integer>();

<http://tutorials.jenkov.com/java-concurrency/read-write-locks.html>

### ReentrantReadWriteLock

The fairness of the waiting thread : approximately arrival-order policy

## Mutex

Yet to start

## Queues used in Threads

### Blocking Queue

Threads will wait () until the enqueue or dequeue operation is performable. That is until queue has lessThanLimit .

The waiting thread will get notifyAll() to perform their operation.

notifyAll() will be called on only one condition in each enqueue and dequeue

#### Synchronous Queue

* A synchronous queue does not have any internal capacity, not even a capacity of one.
* Cannot peek at a synchronous queue because an element is only present when you try to remove it.
* Cannot insert an element (using any method) unless another thread is trying to remove it
* Cannot iterate as there is nothing to iterate
* Does not permit null elements.
* By default, ordering of waiting producer and consumer threads is not guaranteed.

#### PriorityBlockingQueue

#### ArrayBlockingQueue

#### LinkedBlockingQueue

Sadasd

## Executor FrameWork

### Thread Pools

A thread pool is a group of threads initially created that waits for tasks to be assigned for execution. After the execution of the current task, it returns to the pool. Hence, reducing the performance overhead associated with starting a new thread and allocating some memory in stack.

The idea is to have the threads always existing, so that we won't have to pay overhead time for creating and managing them every time. The number of threads running in your application at a time can be handled by thread pool.

Generally, a Java thread pool is composed of:

* the pool of worker threads, responsible for managing the threads(Executor Interface)
* a thread factory that is responsible for creating new threads( Executor**S** class)
* a queue of tasks waiting to be executed.----------- isn’t it part of the executor framework ?

The **Executor framework** uses the concept of threadPool and takes the overhead of creating and managing the threads of the pool. Instead of creating multiple threads just create one Executor and execute the tasks(Runnable)

### Executors class

Executors class contains factory methods for creating different types of thread pools.

**Types of Executors**:

There are different types of Executors(Thread Factories) to create a Thread Pool.

#### SingleThreadExecutor

Tasks are places in a LinkedBlockingQueue

#### FixedThreadPool

If the tasks are more than the available threads, they are stored in LinkedBlockingQueue.

#### CachedThreadPool

Synchronous Queue.The queue size will always be zero.

#### ScheduledExecutor

DelayedWorkQueue

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

### Executor Interface

Executor interface and its subInterfaces/subclasses are meant for executing, managing, and scheduling the execution of tasks.

The only method of Executor Interface is execute() which takes an Runnable and returns void.

#### ExecutorService

Besides the execute() method, this interface also defines a similar submit() method that can return a Future object.

This service should be explicitly shutdown by shutdown() or shutdownow()

**shutdown**, initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

**shutdownow**, attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

**awaitTermination**(long, TimeUnit), Blocks until all tasks have completed execution. Returns true if this executor terminated and false if the timeout elapsed before termination.

So, if false is returned it is better to call shutdown() or shutdownNow().With this approach, the ExecutorService will first stop taking new tasks and then wait up to a specified period of time for all tasks to be completed. If that time expires, the execution is stopped immediately.

##### ScheduledExecutorService

The *schedule()* method specifies a task to be executed, a delay value and a *TimeUnit* for the value:

Future<Double> future = executor.schedule(callableTask, 2, TimeUnit.MILLISECONDS);

Furthermore, the interface defines two additional methods:

scheduleAtFixedRate( () -> print("Fixed Rate Scheduled"), 2, 2000, TimeUnit.MILLISECONDS);

scheduleWithFixedDelay( () -> print("Fixed Delay Scheduled"), 2, 2000, TimeUnit.MILLISECONDS);

The scheduleAtFixedRate() method executes the task after 2 ms delay, then repeats it at every 2 seconds. Similarly, the scheduleWithFixedDelay() method starts the first execution after 2 ms, then repeats the task 2 seconds after the previous execution ends.

##### ThreadPoolExecutor

The **corePoolSize** parameter is the number of core threads that will be instantiated and kept in the pool. When a new task comes in, if all core threads are busy and the internal queue is full, the pool is allowed to grow up to **maximumPoolSize**.

The **keepAliveTime** parameter is the interval of time for which the excessive threads (instantiated more than the corePoolSize) are allowed to exist in the idle state. By default, the ThreadPoolExecutor only considers non-core threads for removal. To apply the same removal policy to core threads, we can use the allowCoreThreadTimeOut(true) method.

##### ForkJoinPool

Instead of single task queue, each thread in the ForkJoinPool has its own double-ended queue which stores tasks.

**Work Stealing Algorithm**:

By default, a worker thread gets tasks from the head of its own deque. When it is empty, the thread takes a task from the tail of the deque of another busy thread or from the global entry queue, since this is where the biggest pieces of work are likely to be located.

ForkJoinTask is the base type for tasks executed inside ForkJoinPool. In practice, one of its two subclasses should be extended: the RecursiveAction for void tasks and the RecursiveTask<V> for tasks that return a value. They both have an abstract method compute() in which the task’s logic is defined.

**Methods:**

execute(Runnable) :

submit(Runnable) : future.get();

submit(Callable) : future.get(); Object call() throws Exception

invokeAny(...) : If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.

invokeAll(...) : Returns a collection of Future objects. Future.get()

### Future Object

Future object functions as a handle to the result of the asynchronous task. Once the asynchronous task completes, the result can be accessed via the Future object returned when the task was STARTED.

Concrete implementation of Future Interface is FutureTask

public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning)

V get(); //get() method will **block until** the result is ready.

V get(long timeout, TimeUnit unit);

boolean isCancelled();

boolean isDone();

}

### Callable Object

Since Callable returns a Value and can also throw Exceptions, Callable are preferred instead of Runnable when assigning tasks to the Thread/Executors.

How Callable is replacing Runnable:

Assign the Callable to the constructor of FutureTask which is Runnable. The FutureTask run() is calling the call() of Callable. The return value or Exception from the Call() can be retrieved from the FutureTask’s get().

## Synchronizers

### CountDownLatch

The CountDownLatch class has three basic elements:

The initialization value that determines how many events the CountDownLatch class waits for. The await() method, called by the threads that wait for the finalization of all the events. The countDown() method, called by the events when they finish their execution.

When you create a CountDownLatch object, the object uses the constructor's parameter to initialize an internal counter. Every time a thread calls the countDown() method, the CountDownLatch object decrements the internal counter by one unit. When the internal counter arrives to 0, the CountDownLatch object wakes up all the threads that were waiting in the await() method.

### CyclicBarrier

CyclicBarrier(int threadsToWait,Runnable barrierAction) : Creates a CyclicBarrier object with the number of threads waiting on it specified. you can pass an additional Runnable object as an initialization parameter, and the CyclicBarrier class executes this object as a thread when all the threads have arrived at the common point.

int await() : Blocks until the specified number of threads have called await() on this barrier.

Cyclic barrier is re-used by reset(), which resets the threadsToWait .

The isbroken(), returns true if any of the waiting threads are interrupted.

CountDownLatch cannot be used.

### Semaphores

The permit machine.

The fairness in the waking up of the blocked threads can be achieved by passing a True value as the second argument of the Semaphore constructor.

The additional argument of the acquire(), acquireUninterruptibly(), tryAcquire(), and release() methods acquires the given number of permits from this semaphore, blocks itself until all are available or the thread is interrupted.

### Exchanger

Exchangers is more like an object for communication than synchronization. Used between a pair of threads.

Exchangers don't break like barriers can (not even when timeouts and interrupts occur).

Exchanger<String> ex = new Exchanger<String>().

The difference between Exchanger and SynchronousQueue is that the producer is blocked until a consumer is waiting for the product. The SynchronousQueue is like of size zero.

### Phaser

The number of threads registered to synchronize on a phaser may vary over time.

The Parties can register even after the construction of the Phaser object. Likewise, parties can arrive and deregister.