Contents

[Read these web pages 5](#_Toc201840380)

[Topics to Learn 6](#_Toc201840381)

[Topic Finisher 6](#_Toc201840382)

[Git 6](#_Toc201840383)

[Git Bash 7](#_Toc201840384)

[Setup 7](#_Toc201840385)

[Checkout 8](#_Toc201840386)

[Branch 8](#_Toc201840387)

[Push 8](#_Toc201840388)

[Reset 9](#_Toc201840389)

[Fetch 9](#_Toc201840390)

[Rebase 10](#_Toc201840391)

[Question 10](#_Toc201840392)

[Puzzle 11](#_Toc201840393)

[Code These 11](#_Toc201840394)

[Code Review Checklist 12](#_Toc201840395)

[Cool Concepts 12](#_Toc201840396)

[General 13](#_Toc201840397)

[Java 13](#_Toc201840398)

[Robust and Secure 13](#_Toc201840399)

[Architecture Neutral 13](#_Toc201840400)

[Portable 13](#_Toc201840401)

[Platform 14](#_Toc201840402)

[Compilation 14](#_Toc201840403)

[JVM 15](#_Toc201840404)

[Java Native Interface 15](#_Toc201840405)

[JRE 15](#_Toc201840406)

[JDK 15](#_Toc201840407)

[Java Architecture 15](#_Toc201840408)

[JVM Architecture 16](#_Toc201840409)

[Class Loader 17](#_Toc201840410)

[Loading 17](#_Toc201840411)

[Linking 17](#_Toc201840412)

[Initialization 18](#_Toc201840413)

[Runtime Memory Area / Runtime Data Area 18](#_Toc201840414)

[Method Area 19](#_Toc201840415)

[Heap Area 19](#_Toc201840416)

[Stack Area 19](#_Toc201840417)

[Program Counter (PC) Registers 19](#_Toc201840418)

[Native Method Stacks: 19](#_Toc201840419)

[Execution Engine 19](#_Toc201840420)

[Interpreter 19](#_Toc201840421)

[JIT Compiler 19](#_Toc201840422)

[Garbage Collector 20](#_Toc201840423)

[Software Architecture 20](#_Toc201840424)

[Definition 20](#_Toc201840425)

[Concepts 24](#_Toc201840426)

[Instructions Reordering 24](#_Toc201840427)

[Pass by value or Reference 24](#_Toc201840428)

[Class class 24](#_Toc201840429)

[Object Class 25](#_Toc201840430)

[Reflection 25](#_Toc201840431)

[Declared or effectively final 27](#_Toc201840432)

[For Loop 27](#_Toc201840433)

[Switch Statement 28](#_Toc201840434)

[Functional & Object Oriented 28](#_Toc201840435)

[Lambda expression 28](#_Toc201840436)

[Default Methods 28](#_Toc201840437)

[Functional Interface 29](#_Toc201840438)

[Streams 30](#_Toc201840439)

[Recursion 32](#_Toc201840440)

[Memory 32](#_Toc201840441)

[Strings 32](#_Toc201840442)

[Regular Expression 32](#_Toc201840443)

[Cloning / Copying 34](#_Toc201840444)

[Immutable 35](#_Toc201840445)

[Exception 35](#_Toc201840446)

[Enhanced Try Block 37](#_Toc201840447)

[Class Not Found 37](#_Toc201840448)

[Custom Exception 38](#_Toc201840449)

[Enumeration 38](#_Toc201840450)

[Serialization 38](#_Toc201840451)

[Externalization 40](#_Toc201840452)

[Reference 40](#_Toc201840453)

[Strong 40](#_Toc201840454)

[Soft 40](#_Toc201840455)

[Weak 41](#_Toc201840456)

[Phantom 41](#_Toc201840457)

[Garbage Collection 41](#_Toc201840458)

[Types of Garbage Collectors 43](#_Toc201840459)

[Serial GC 43](#_Toc201840460)

[Parallel GC 44](#_Toc201840461)

[Concurrent Mark & Sweep (CMS) GC 44](#_Toc201840462)

[Garbage First GC 45](#_Toc201840463)

[Compare Objects 45](#_Toc201840464)

[Comparable 45](#_Toc201840465)

[Comparator 45](#_Toc201840466)

[Data Structures 46](#_Toc201840467)

[Binary Tree 46](#_Toc201840468)

[Linked List 46](#_Toc201840469)

[Sorting 46](#_Toc201840470)

[HeapSort 46](#_Toc201840471)

[Bubble Sort 47](#_Toc201840472)

[Collection 47](#_Toc201840473)

[Concepts 47](#_Toc201840474)

[HashCode & Equals 47](#_Toc201840475)

[Iterable 48](#_Toc201840476)

[Iterator 48](#_Toc201840477)

[ListIterator 48](#_Toc201840478)

[Collections Class 48](#_Toc201840479)

[Collection 48](#_Toc201840480)

[List 48](#_Toc201840481)

[Queue 49](#_Toc201840482)

[Map 50](#_Toc201840483)

[Set 53](#_Toc201840484)

[Complexity 54](#_Toc201840485)

[Standard Algorithms 54](#_Toc201840486)

[Order of Complexity 54](#_Toc201840487)

[Design Pattern 56](#_Toc201840488)

[Behavioural 56](#_Toc201840489)

[Chain of Responsibility 56](#_Toc201840490)

[Strategy 56](#_Toc201840491)

[Observer 57](#_Toc201840492)

[Command Pattern 57](#_Toc201840493)

[Iterator Pattern 58](#_Toc201840494)

[Template Method 60](#_Toc201840495)

[State Pattern 60](#_Toc201840496)

[Structural 61](#_Toc201840497)

[Decorator Pattern 62](#_Toc201840498)

[Adapter Pattern 64](#_Toc201840499)

[Bridge Pattern 65](#_Toc201840500)

[Façade 65](#_Toc201840501)

[Proxy Pattern 66](#_Toc201840502)

[Composite 66](#_Toc201840503)

[FlyWeight 66](#_Toc201840504)

[Creational 66](#_Toc201840505)

[Static Factory 66](#_Toc201840506)

[Factory Method 66](#_Toc201840507)

[Abstract Factory 67](#_Toc201840508)

[Singleton 67](#_Toc201840509)

[Builder Pattern 71](#_Toc201840510)

[Prototype 71](#_Toc201840511)

[Design Principles 72](#_Toc201840512)

[Single responsibility 72](#_Toc201840513)

[Open-Closed Principle 72](#_Toc201840514)

[Liskov Substitution Principle 72](#_Toc201840515)

[Interface Segregation Principle 73](#_Toc201840516)

[Dependency Inversion Principle 73](#_Toc201840517)

[Threads 74](#_Toc201840518)

[Simple Definitions 74](#_Toc201840519)

[Cost of creating & starting Threads 75](#_Toc201840520)

[Daemon Thread 75](#_Toc201840521)

[Basic wait & notify 75](#_Toc201840522)

[Interrupt 76](#_Toc201840523)

[Thread Safety 76](#_Toc201840524)

[Volatile 76](#_Toc201840525)

[Happens-Before Guarantee 77](#_Toc201840526)

[Atomic Objects 78](#_Toc201840527)

[Blocking & Non-blocking algorithms 78](#_Toc201840528)

[Intrinsic Lock or Monitor Lock 78](#_Toc201840529)

[Synchronize 79](#_Toc201840530)

[Lock Interface 82](#_Toc201840531)

[ReEntrantLock 82](#_Toc201840532)

[Spin Lock 82](#_Toc201840533)

[ReadWriteLock Interface 82](#_Toc201840534)

[ReentrantReadWriteLock 82](#_Toc201840535)

[Callable, FutureTask 83](#_Toc201840536)

[Queues in Threads 83](#_Toc201840537)

[Blocking Queue 83](#_Toc201840538)

[Executor Framework 85](#_Toc201840539)

[Thread Pools 85](#_Toc201840540)

[Executor Interface 85](#_Toc201840541)

[Synchronizers 90](#_Toc201840542)

[CountDownLatch 90](#_Toc201840543)

[CyclicBarrier 90](#_Toc201840544)

[Semaphores 90](#_Toc201840545)

[Mutex 90](#_Toc201840546)

[Exchanger 91](#_Toc201840547)

[Phaser 91](#_Toc201840548)

# Read these web pages

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

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

Git is a version control system. It is a command-line tool (and set of libraries) that lets you:

1. Track changes in code or text files.
2. Collaborate with others via branching, merging, pushing, pulling, etc.
3. Revert to previous versions.
4. Manage project history efficiently.

## Git Bash

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. (Git bash = Git + Bash on **windows**)

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.

## Setup

git config --global user.name "JamesTharakan"

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

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

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](https://github.com/%3cname%3e/%3crepoName%3e.git)

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

origin is a remote repository name. It is used as a convenient alias for the URL of the remote repository, allowing you to interact with it without typing out the full URL each time.

**HEAD** refers to the tip/unPushedCommit(s) of the current branch you are on.

Example: git push origin HEAD:remote\_branch

Head: It is like a pointer to the tip off the current branch.

## Checkout

## Branch

First, you need to create your branch locally and checkout that branch to work locaaly.

git branch yourBranch

git check yourBranch

or

git checkout -b your\_branch

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

$ git branch -d branch\_name //delete

$ git branch -D branch\_name

$ git push origin --delete <branch\_name> //deletes the remote branch

## Push

The command pushes the commits in the current branch to the remote and tracks it.

**git push origin HEAD:master**

**git push origin HEAD:remote\_branch**

**i.e**

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: remote \_branch** //this will push the latest commit to the remote branch

**git push origin your\_branch : remote\_branch**

**git push origin master or git push origin master:master**

This command pushes the commits in the **your\_branch** to **remote\_branch** in the remote. That is, you don’t have to be in the your\_branch to do this.

Review system:

git push origin HEAD:refs/for/evo\_main

The prefix refs/for/ is a special Gerrit namespace. Gerrit-specific push — not standard Git.

It does not update the actual remotebranch directly.Instead, it creates a change request (like a pull request / merge request) in Gerrit.Gerrit stores these in a hidden ref (refs/changes/...) and lets reviewers approve/reject them.

## Reset

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>

## Fetch

git fetch origin master

the above command does 2 steps

1. Fetch Updates: Git retrieves the latest commits, files, and references from the remote repository's master branch.
2. Update Remote-Tracking Branch: The fetched changes are stored in your local repository under the remote-tracking branch named origin/master.

Now we can, After fetching, you can compare your current branch with origin/master

Git diff origin/master

After inspecting, to merge the latest changes from origin/master into your current branch, you can use: git merge origin/master

## Rebase

git checkout feature-branch

git rebase main

In this example, the commits from feature-branch are **reapplied** on top of the latest commits from main.

When performing a rebase, conflicts may arise if there are changes in both branches that affect the same lines of code. Git will pause the rebase and allow you to resolve conflicts. After resolving the conflicts, you can continue the rebase with:

git add .

git rebase –continue

Common practice of rebase is 2 steps :

git fetch origin/main

git rebase origin/main

It is important to first fetch the new changes form the repository to the local remote tracking branch. And then rebase

If I simply run the rebase command without fetching, "git rebase origin/main", Git merges the my current branch with the last know status/commitHistory of origin/main. which could be outdated.

# 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

* Build Amazon filters, builder pattern.

# Code These

Use these in your regular programming:

* Streams and its methods
* Enhanced for loop and forEach loop
* Lambda and Functional Interface
* Predefined Functional Interface
* yield and multipleCase in Switch Case

Puzzles:

* Recursion
* Palindrome
* Pallindrome Subsets
* searching in Binary Search

Design Patterns:

* Abstract Factory Pattern
* Iterator
* State: Game. Make the character to walk, eat, talk, run, fight. Pass command and change the state. Change state A-> B, A->A
* Decorator
* Bridge

Algorithms:

* 2 pointer algorithms
* 3 pointer algorithms
* Sliding Window algorithm

TimePass:

* Update all my existing code with Streams APIs.
* Move all methods of Checker class to individual class for better clarity

# Code Review Checklist

Am I able to understand the code easily? Like avoid nested ternary operators, combine/nest in a single statement.

Is the code written following the coding standards/guidelines?

Java Docs or simple one-or-two lines of comment.

Is the same code duplicated more than twice? Make use of Util classes

Can I unit test / debug the code easily to find the root cause?

Is this function or class too big? If yes, is the function or class having too many responsibilities?

Step back and see if this is the right approach to fix this bug.

# 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>"

Some of the important Java 8 features are:

1. forEach() method in Iterable interface
2. default and static methods in Interfaces
3. Functional Interfaces and Lambda Expressions
4. Java Stream API for Bulk Data Operations on Collections
5. Java Time API
6. Collection API improvements
7. Concurrency API improvements
8. Java IO improvements

# General

Why to set the Environment variable?

**Path**: To tell the computer/windows what is java, javac, jar (things under JDK/bin) commands are?

**CLASSPATH**: path of where the user files are placed. Like the \*.class files.

Is Java Compiled or interpreted?

It is both. Compiler compiles and converts it into byte code and then the JVM interprets it at runtime.

When trying to loop in an array we can avoid the end point checks by using:

**Next** : (current+1) % N

**Previous**: (current + N-1) % N

Base or exit condition of recursive method: Smallest Valid input.

# 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 bytecode (binary representation of our code) — 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

Diagram

Description automatically generated

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

Interpreter, C1 and C2 compilers converts the byte code to machine code.

When running the application, the interpreter, 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.

All the above things happens when the application is running, that why they are called JIT compilers.

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.

## JVM

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

JVM is platform **dependent**.

If you have a JRE or JDK, then the simple way to start a JVM is to run the ‘java’ command.

>java helloWorld

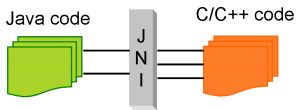
On a typical OS, each JVM runs as a process. Given that the OS allows you to launch multiple processes, you can run multiple JVMs. (This is certainly true for Windows, Linux, MacOSX and other varieties of UNIX on which Java runs.)

So, we can have any number of JVM running in your machine, even with different JDK versions.

Each JVM is a separate process, and they do not share stacks or heaps. Only common core JVM and native libraries will be shared.

Java Native Interface

Java Native Interface (JNI) is a programming framework that allows Java code to interact with native applications and libraries written in other languages.



## 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

It 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.

JDK = JRE (which in turn has JVM) + compiler + Debugger +

## Java Architecture

Explanation

* Have 3 major components, JDK, JRE and JVM.
* JDK: Writing and Compiling Code.
* JRE: Provides all the resources needed to **run** Java applications. This includes the JVM itself along with class libraries, loaders, and other files that support runtime execution of Java programs.
* JVM: The JVM is the **actual engine** that runs the Java bytecode.

**Execution Process**

Run "java MyApp" from the command line. (java launcher is a executable component part of JRE kit)

The operating system runs the java executable of java.exe.

The Java launcher checks the system's environment variables (JAVA\_HOME, PATH) to find the JRE installation.

Once the JRE is located, the Java launcher loads the JVM binary (e.g., jvm.dll) into memory.

After loading the JVM, the launcher initializes the JVM’s components like class loader, bytecode verifier, execution engine.

Then the launcher passes control to the JVM, which begins the process of loading the specified class.

Once the class is loaded and verified, the JVM will invoke the main method of the class, which is the entry point for the application.

# JVM Architecture

A picture containing text, screenshot, diagram, line

Description automatically generated

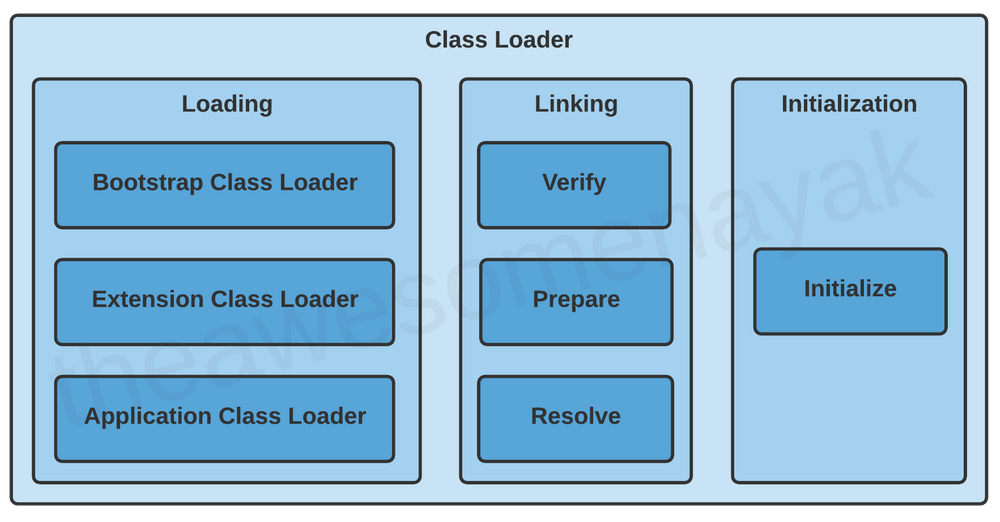
The JVM consists of three distinct components:

* Class Loader
* Runtime Memory/Data Area
* Execution Engine.

## Class Loader

When required by the application, the JVM uses the ClassLoader.loadClass(“fullyQualifiedName”) method for loading Java class dynamically during runtime into the JVM.

There are three phases in the class loading process: Loading, Linking and Initialization.



### Loading

By using Class.getClassLoader() we can find which loader class is used to load a given Class.

**Bootstrap class loader**:

A Bootstrap ClassLoader is a Machine code which starts the operation when the JVM calls it. It is not a java class. Its job is to load the first pure Java Class (may be java.lang.Object). Bootstrap ClassLoader loads classes from the runtime jar (JDK/JRE/LIB, rt.jar). Bootstrap ClassLoader doesn’t have any parent ClassLoader. It is also called as the Primordial ClassLoader.

**Extension class loader**:

The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from jre/lib/ext directory, or any other directory pointed by the system property java.ext.dirs.

**Application or System loader**:

It loads the Application type classes found in the environment variable CLASSPATH, -classpath or -cp command line option. The Application ClassLoader is a child class of Extension ClassLoader.

When requested to find a class or resource, the below steps are followed (Delegation Hierarchy Algorithm):

* A ClassLoader instance checks if the class was already loaded.
* If not loaded, it delegates the search for the class or resource to its parent class loader before attempting to find the class or resource itself.
* If parent class loader cannot load class, it attempts to load the class or resource by itself.

A loaded class is only visible to current loader and its child.

### Linking

This operation combines different files in the main program together. **VPR**

**Verification**

This phase checks the structural correctness of the .class file by checking it against a set of constraints or rules. It also verifies the generation is done by valid compiler or not (What is valid compiler? same version as JRE?). If verification fails for some reason, we get a VerifyException.

**Preparation**

JVM allocates memory for class variables and initializes the memory to default values based on the type.

**Resolution**

The JVM converts these symbolic references into direct references, which are actual memory locations or pointers that the JVM uses to access classes, methods, and fields during execution.

Once symbolic references are resolved into direct references, the JVM knows exactly where in memory to find the class, method, or field, and it can access them directly without any further lookup.

The JVM performs checks during resolution to ensure that all references are valid and adhere to Java’s access control rules (e.g., private, protected, etc.).

### Initialization

All static variables will be assigned with the actual values, and the static block will be executed.

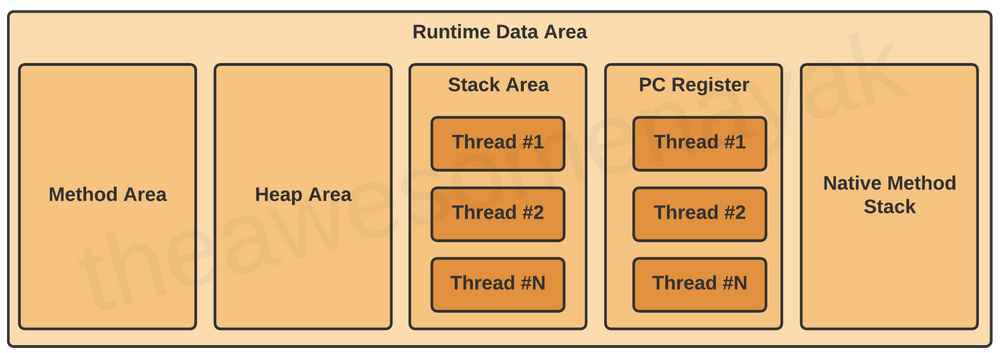
Lazy initialization will be done i.e., only when the class is referred. Following are the initialization reasons:

* When it is a super class of a class whose object is created or when static methods are called.
* Static methods or fields are accessed.
* An instance is created.

## Runtime Memory Area / Runtime Data Area

When we start a program, a Method Area and Heap Area are allocated to the JVM. When the JVM is executing this program, for each method call (by a single or multiple thread) a frame is created in the Stack Area. For every non-native method the next instruction of that method is stored in the PC Registers. That is, each non-native method will have its own PC register.

The native method does not hold any PC Registers.



### Method Area

All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.

### Heap Area

### Stack Area

### Program Counter (PC) Registers

It stores the address of the currently executing JVM instruction.

### Native Method Stacks:

Each thread has its own native method stack. The native method stack is used for executing native (non-Java) code. It holds information about the execution of native methods through JNI.

## Execution Engine

Once the bytecode has been loaded into the main memory, and details are available in the runtime data area (and the class Object is created by JVM), before executing the program, the bytecode needs to be converted into machine language instructions. The JVM can use an interpreter or a JIT compiler for the execution engine.

Read Compilation Section under Java.

### Interpreter

The interpreter reads and executes the bytecode instructions line by line. Due to the line-by-line execution, the interpreter is comparatively slower. It uses map/dictionary to map the bytecode to machine language. Another disadvantage of the interpreter is that when a method is called multiple times, every time a new interpretation is required.

### JIT Compiler

The execution engine first uses the interpreter to execute the bytecode line-by-line and it will use the JIT compiler when it finds some repeated code. (Eg: calling the same method multiple times, accessing memory in for loop). At that time JIT compiler compiles the entire bytecode into native code (machine code). These converted native codes will be stored in the cache. So, whenever the repeated method is called, this will provide the native code. Since the execution with the native code is quicker than interpreting the instruction, the performance will be improved.

**Intermediate Code Generator** – Produces intermediate code

**Code Optimizer** – Responsible for optimizing the intermediate code generated above

**Target Code Generator** – Responsible for Generating Machine Code or Native Code

**Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not

### Garbage Collector

Refer Garbage Collector topic

# 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.

# 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 JVM** or **your code** (if you explicitly throw an exception) is responsible for creating the **exception object**. It 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 in the system is called throwing an exception.

If no method in the call stack catches the exception, then the exception reaches the JVM’s runtime system, which will:

* Print the stack trace.
* Terminate the program with an uncaught 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**

Defines a contract for classes to implement. It specifies a set of abstract methods (method signatures without bodies) that a class must implement if it chooses to "implement" the 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.

* Separate how we use something from how it is implemented.
* 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.
* Interfaces that completely describe the functionality of a class are usually wrong
* Interfaces are always implemented by more than one class.

**Abstract class**

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

Can’t 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**

Inheritance allows a subclass to acquire the characteristics (attributes and methods) of its superclass, enabling the subclass to use and extend the functionality of the superclass.

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.

**Polymorphism**

Static polymorphism (Compile time Polymorphism)

The object binding with method is decided at compile-time based on the parameters or arguments of method. Method overloading

Dynamic polymorphism (Runtime Polymorphism):

A polymorphism where object binding with methods happens at runtime is called runtime polymorphism. Method overriding.

**Abstraction**

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

Abstraction can be achieved by using interfaces and abstract classes.

**Encapsulation**

Encapsulation is a process of binding the data and the methods that operates on the data into a single unit, typically called a **class**.

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

**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 straight line. Both relation belong to “has-a” relation.

**Composition**

Composition: contains

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. Round Robin.

**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

## 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>

# Class class

Text

Description automatically generated

Every different type of object in the running Java application will have a single Class object. These Class objects are created by the JVM. They don’t have any public constructor. We can fetch that Class object using Class.forname("fullyQaulifiedName"). When we write someClass.class, it references to this Class object.

Class object is constructed automatically by the Java Virtual Machine when a class loader invokes one of the defineClass methods and passes the bytes of a class file (byte code).

The Class object also represents enums and intefaces besides classes in a running Java application and has their metadata. The Class object is mostly used with the reflection capabilities of Java.

The Class object has metadata about the class. Some for the information are:

* Package
* Fully qualified class Name
* Parent class name
* Fields
* Constructors
* Annotations
* Methods
* Constant Pool

With this class Object we cannot call any methods of the Class.

# Object Class

In Java we can synchronize the access to the critical section of the code using **any** Object, not just the Thread objects. That’s why the methods wait,notify and notifyAll are part of Object class and not in Thread class.

Example: One Chair, 3 threads of a single Person class. If

# Reflection

Type introspection is the process of inspecting or examining metadata about a class during runtime. It allows you to gather information such as:

* Class name.
* Fields.
* Constructors.
* Methods.
* Modifiers (public, private, static, etc.).
* Interfaces and superclass.

Type introspection is done by using the information available in the Class object. Class.getDeclaredFields(), Class.getDeclaredMethods(), and Class.getConstructors().

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

While introspection is focused on gathering information, reflection allows modifying, accessing, and invoking class members at runtime.

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

**With reflection we can**:

-create new instances of a class and calling methods at runtime, even if the class name is only known at runtime. Constructor.newInstance(), Method.invoke().

-access private or protected members of a class.Field.setAccessible(true) and Field.set().

-to inspect the annotations applied to classes, methods, and fields. and make decisions dynamically at runtime.

Class Object, Field Object, Method Object

**Use cases**:

- Many Java frameworks (e.g., Spring, Hibernate) use reflection internally to inspect classes and dynamically perform actions, like dependency injection or database mapping, without requiring explicit user intervention.

- to serialize objects into formats like JSON or XML and to deserialize them back into Java objects.

- JUnit use reflection to invoke test methods dynamically.

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

**Example**:

// Invoke the private method 'secretMethod'

Method secretMethod = exampleClass.getDeclaredMethod("secretMethod");

secretMethod.setAccessible(true); // Bypass private access

secretMethod.invoke(example); // Output: Secret message!

Class.forName("Foobar").getDeclaredConstructor().newInstance()

Constructor constructor = Class.forName("java.lang.String").getConstructor(String.class);

String object = (String) constructor.newInstance("Hello");

**Blocking Reflection**:

Extend the SecurityManager class and override this method to restrict reflection access to specific package. Also, the SecurityManager has to be configured at the JVM level.

So, in the below example we are restricting the access to the package: java.lang.reflect

public void checkPackageAccess(String pkg){

super.checkPackageAccess(pkg) //This line is a MUST HAVE CODE.

if(pkg.equals("java.lang.reflect")){

throw new SecurityException("Reflection is not allowed!");

}

}

# 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 prohibits 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.

# For Loop

**For Loop**:

for (data-type variable; condition; increment) {

// Code to be executed

}

**Enhanced For Loop**:

For (data-type variable: array or collection) {

// Code to be executed

}

**For Each**:

list.forEach(Consumer);

**Drawbacks** of enhanced for loop or for Each:

* We cannot track the index if these used.
* In case of primitives, we cannot modify the current item in the original array.
* They can only iterate forward in single steps and cannot jump. No reverse order.
* **Enhanced for-loops** and **forEach()** are convenient for iterating over elements, but they cannot be used for **modifying elements** in the original array or collection.

# Switch Statement

Yeid and -> operator

# Functional & Object Oriented

Emphasizes the evaluation of functions and avoids changing state and mutable data. It focuses on what to compute rather than how to compute it.

# Lambda expression

(parameters) -> {Body}

parameter - > single statement with implicit return (if required).

The shortest form of implementing the Interface.

**Usage**:

1. Functional interfaces and lambda expressions work together to provide a way of passing behaviour (a block of code) in Java.
2. Collections: forEach.
3. In stream APIs: filter, maps, reduce. [Which other Stream APIs it can be used.]

Compiler does Type Inference to find what type of lambda/data it is.

# 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.

When we extend an interface that contains a default method, we can override or not override the default method. i.e., default methods can be overridden.

If a class inherits a default method from multiple interfaces, it must explicitly override the method to resolve the conflict.

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

Single purpose interface.

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

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

Functional interfaces can be annotated with @FunctionalInterface which will make sure there will be compiler error if more than one abstract method is accidentally added.

Functional interfaces and lambda expressions work together to provide a way of passing behaviour (a block of code) in Java.

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

Based on arguments and return:

**Consumer**: Used to define a behaviour that takes single argument through accept method and should return void. This interface can be used to log, do some prerequisite steps & trigger event, wrap an object & propagate etc.

**Predicate**: Single argument, should returns Boolean. Extensively used as an input parameter in the filter () of Stream API.

Predicate <String> tester = s -> s.equals("pass");

System.out.println(tester.test("yes"));

We can combine predicate too.

Predicate isEven = x -> x % 2 == 0;

Predicate isGreaterThanThree = x -> x > 3;

Predicate combined = isEven.and(isGreaterThanThree);

System.out.println(combined.test(4));

**IntPredicate:** test () which takes a single int value argument and returns a boolean.

IntPredicate isPresentInDB = x -> true; // A short and simple code to check if an ID is in DB

System.out.println(isPresentInDB.test(2408));

we can combine them too.

**Supplier**: No Input but returns an object. get ()

Usecases:

* Generate values or retrieve the next available values
* Create a new instance when it required or may be computationally expensive. Lazy.
* Provide a default value when something is null.

No, it's not meant to improve performance. The Supplier is used for a deferred execution i.e. you specify a functionality (code) that will run whenever used.

**Function**: Single argument, returns Object.

**Bifunction**: Two argument, returns Object.

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.

# 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.

Each stream can be used only once. That is the stream can be consumed only once.

In the parallel Stream we can launch multiple threads on the different parts of the Stream and then combine the result.

Code using Stream majorly has 3 parts: Source, zero or more intermediate operations and a terminal operation

**Intermediate Operation**

They usually accept functional interfaces as parameters Return a new stream that represents the result of the intermediate operation.

**Terminal Operation**

These operations are applied to a stream to produce a final result or trigger the processing of the stream. Once a terminal operation is invoked on a stream, the stream is consumed and the stream cannot be used again.

**Create**: range(), rangeClosed(), stream(),parallel(), parallelStream(),sequential()

**mapping**: map(), flatMap(), mapToInt(), mapToLong(), mapToDouble(),mapToObj()

**Filtering**: filter(), distinct()

**Sorting**: sorted(), sorted(Comparator)

**Limiting and Skipping**: limit(), skip()

**Peeking and Debugging**: peek()

**Matching**: anyMatch(), allMatch(), noneMatch()

**Finding**: findAny(), findFirst()

**Reduction**: reduce()

**Short-Circuiting**: takeWhile(), dropWhile()

**Concatenation or Combining**: concat()

**Colletions**:collect(), toList(),toSet(),toMap(),toCollection(Supplier<C>collectionFactory),collectingAndThen()

**Aggregation Operations**: counting(),minBy(),maxBy(),Collectors.summingInt(ToIntFunction Mapper),Collectors.averagingInt(ToIntFunction Mapper),joining(),summarizingInt(ToIntFunction Mapper)

**Grouping Operations** :partitioningBy(),groupingBy

**Numeric operation**: sum(), average(), min(), max()

**filter**(Predicate<T> predicate) — accepts a predicate to filter all elements of the stream.

**map**(Function<T,R> mapper)-Transforms stream to a stream consisting of the results of applying the given function to the elements of the stream.

**sorted**(), sorted(Comparator<T> comparator) — elements in the stream are sorted in natural order unless we pass a custom Comparator.

**flatMap**(Function<T, Stream <R>> mapper)-flatten the objects from all the collections in the original Stream into a single collection.

**distinct**()-returns a stream consisting of the distinct elements.

**peek**(Consumer<T> action)-returns a new Stream consisting of all the elements from the original Stream after applying a given Consumer action. As per javadoc this method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline.

**limit**(long maxSize)-is short-circuiting intermediate operation, returns a new stream consisting of the elements of the given stream, truncated to be no longer than maxSize in length.

**skip**(long n) — returns a stream consisting of the remaining elements of the stream after discarding the first n elements of the stream.

**forEach**(Consumer<T> action) — performs specified action for each element of this stream.

**forEachOrdered**(Consumer<T> action) — similar to forEach, but it guarantees that elements are processed in the order of the original stream, even for parallel streams.

**toArray**() — returns an array containing the elements of the given stream.

**collect**(Collector<T, A, R> collector) —collects the elements of the stream into a collection or a custom aggregation.

**reduce**(T identity, BinaryOperator<T> accumulator) — produce a single result by repeatedly applying a function(binary operator) to a sequence of elements from a stream.

**count**() -returns stream’s size.

**min**(Comparator<T> comparator) — returns the minimum element of this stream according to the provided Comparator.

**max**(Comparator<T> comparator) — returns the maximum element of this stream according to the provided Comparator.

**anyMatch**(Predicate<T> predicate) — checks if any element matches a predicate.

**allMatch**(Predicate<T> predicate) — checks if all elements match a predicate.

**noneMatch**(Predicate<T> predicate) — checks if no element matches a predicate.

**findAny**() — finds any element in the stream.

**findFirst**() — finds the first element in the stream.

**When not to use streams**:

* **Use Loops** when you need more control over the flow of execution, or when performance is critical in very tight loops where streams may add some overhead.
* Loops may also be more suitable for highly complex logic that doesn't fit the declarative style of streams.
* Streams are harder to debug compared to traditional loops. With loops, you can easily print out values at each step or use a debugger to step through the iteration.
* Streams abstract away the iteration, so you don’t have direct access to indices like you do in traditional for loops

# Recursion

**Exit condition**: Look for smallest Valid input.

**Memoization** is a way to optimize DP algorithms which rely on recursion. The point is not to solve the sub problem again which has been already solved. You can view it as cache of solutions to sub problems.

**Dynamic Programming** is a way to solve problems which exhibit a specific structure (optimal sub structure) where a problem can be broken down into sub problems which are similar to original problem. Clearly one can invoke recursion to solve a DP. But it is not necessary. One can solve a DP without recursion.

# Memory

Stack memory only contains local primitive variables and reference variables to objects, whereas the actual Object is 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 bits on 32-bit CPU, 64 bits 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

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 a new Object with following rules:

1. it will create an exact copy of all its primitive .
2. For immutable instance variable of the object, it will copy the reference
3. For mutable instance variable of the object,
   1. if it does not implement cloneable, it will copy the reference.
   2. If it implements cloneable, recursive.

CloneNotSupportedException is thrown to show that the clone method of Object class has been called to clone an object, but the object’s class does not implement the Cloneable interface.

CloneNotSupportedException: clone() method is overridden but it does not implement Cloneable interface.

Error: If the method is also not overridden in the class, then we will get compilation error because the clone method is Protected and is not visible in our class.

**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

// Overriding clone() method to create a deep copy of an object.

protected Object clone() throws CloneNotSupportedException {

Employee emp = (Employee) super.clone();

emp.dept = (Department) dept.clone();

return emp;

}

**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.

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**.

So why do we need cloning when copy constructor is there?

# Immutable

To put it simply, a class instance is immutable when its internal state can't be modified after it has been constructed.

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. Simplicity - each class is in one state only.

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.
* If a field is a mutable object create defensive copies of it for getter methods.
* No Setters. Getters should return a deep copy of instance variable.

Primitives, String, numeric wrapper objects etc are immutable.

what does it mean by "Immutable Objects are thread safe"?

Answer: No matter how many threads are there, it cannot change the actual object.

A new object is created every time someone tries to modify. This makes the Object thread safe. Hence synchronising is not needed.

Mutable objects are not thread safe because multiple threads can write to that data at the same time.

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.

Diagram

Description automatically generated

Exceptions are further divided into checked exceptions and Unchecked Exceptions.

Error class, RuntimeException class and their subclasses ARE UNCHECKED (RED)

**Checked**: These are checked at compile time to ensure we are handling them, either by catching them or declaring the containing method throws the exception.

**Unchecked**: Where the application usually cannot anticipate or recover from. We should not handle.

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. RuntimeException and all its exception subclasses are not checked by Java compiler because they occur during runtime of a program. That’s why these exceptions are also called unchecked exceptions.

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**: IndexOutOfBoundException, ArithmeticException, ClassCastException, IllegalArgumentException, NullPointerException

**Checked Exceptions**:

FileNotFoundException, SQLExecption

While the Java Language Specification does not require it, there is a strong convention that errors are reserved for use by the JVM to indicate resource deficiencies, invariant failures, or other conditions that make it impossible to continue execution.

### Enhanced Try Block

The resources mentioned in the try parenthesis are closed automatically when the try block is exited. Multiple resources can be mentioned in the try block.

You can pass any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable.

**try/finally vs tryWithResources**

The exception which is propagated out of try catch block differs in try and tryWithResource ,

Try{throws e1}finally{throws e2}

try(withResourse){}catch{e1|e2}

**try-catch-finally block** : last exception encountered is the exception that is propagated In .e2 is send down the call stack.

**withResoucre** : semantic the first exception is propagated. First occurred exception is sent down the call stack.

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 and can be shown in the UI besides in the log file.", e); }

### Class Not Found

ClassNotFoundException and NoClassDefFoundError occur when a particular class is not found at runtime and are related to classpath.

**ClassNotFoundException**(in classPath):

Thrown when an application tries to load in a class through its string name using:

* The method in Class. forName()
* The method in class ClassLoader.loadClass().
* The findSystemClass method in class ClassLoader .

Since the String representation can only validated in runtime it will cause exception during running.

Most of the time we get the ClassNotFoundException when the ClassPath is not updated with the required JAR files. Class.forName("oracle.jdbc.driver.OracleDriver");

**NoClassDefFoundError** :

Thrown if the Java Virtual Machine or a ClassLoader instance tries to load in the definition of a class (as part of a normal method call or as part of creating a new instance using the new expression) and no definition of the class could be found.

The searched-for class definition existed when the currently executing class was compiled, but the definition can no longer be found.

When does it come: example?

Just by deleting the dot class file we will not get this ERROR. Deleting the file will only give classNotFoundException as JVM does not realize the existence of such class.-- May be

**Could not find or load main class** : This is when the class that we wanted to run does not have the main method.

### Custom Exception

Simply extend the Exception class and give a single argument constructor. Call the super class in the constructor.

# Enumeration

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

When we compile an enum, the Java compiler does these:

* It turns the enum into a subclass of the abstract class java.lang.Enum, hence custom enums can’t inherit any other class/enum.
* It compiles the enum as a **final** class

# 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.

Static and transient instance members are not saved during serialization. During deserialization the constructors are not called.

**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?

The automatically generated UID is generated based on various aspects of the class( class name, implemented interfaces, and all public and protected members). Changing any of these in any way will change the serialVersionUID.

So, declare an explicit serial version UID in every serializable class, this eliminates the serial version UID as a potential source of incompatibility and unnecessary computation of UID.

Java specification says, "For serializable objects, the no-arg constructor for the first non-serializable supertype is executed. For serializable classes, the fields are initialized to the default value appropriate for its type. Then the fields of each class are restored by calling class-specific readObject methods, or if these are not defined, by calling the defaultReadObject method. Note that field initializers and constructors are not executed for serializable classes during deserialization."

Ex class :Temperature

For creating Temperature object, it will not call constructor of Temperature class instead it uses ReflectionFactory class internally for creating the object, which initializes the fields of Temperature object with default values according to variable type first and later it reinitializes the variables with the value present in deserialized stream to restore back the original value which was present at the time of Serialization.

Deserialization process is, it starts checking which class object state needs to be preserved and for identifying this it looks for class implements Serializable interface and where it first finds a class which doesn't implements Serializable interface, it calls constructor of that class because it doesn't care if value of that class will be overridden because it is already marked non serializable. For class which marked Serializable, it constructs object using ReflectionFactory and restore value of that class from deserializing stream.

**Basic read and write**:

The ObjectOutputStream class contains writeObject() method for serializing an Object.

public final void writeObject(Object obj) throws IOException

The ObjectInputStream class contains readObject() method for deserializing an object.

public final Object readObject() throws IOException,ClassNotFoundException

**Custom overrides**:

**Need** :

* Encrypt/decrypt before serialization.
* execute any specific piece of code before/after serialization.

readObject(ObjectInputStream ois) : method will use this method for reading the object from stream for any custom data reading besides the usual non-transient and non-static data.

writeObject(ObjectOutputStream oos): method will use this method for writing the object to stream for any custom data reading besides the usual non-transient and non-static data. One of the common usages is to obscure the object variables to maintain data integrity.

In both the above methods we must call the defaultWriteObject and defaultReadObject first. Which performs the basic read and write of non-transient and non-static members. This method call also helps in maintaining the backward and future compatibility.

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.

These two methods can be used to block duplicating Singleton using serialization.

Object readResolve() { Singleton s = getInstance(); s.i = i; return s; }

# Externalization

readExternal() and writeExternal()

# 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 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 reference , which can be confirmed by polling the referent queue.

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.

When the collector determines that the referent is softly, weakly, or phantom reachable, the reference object will be enqueued in the registered reference queue.

A program can then poll the reference queue to know when the referent object is reachable only through reference objects. Once found the entry in queue, the program can proceed with clean-up operations on other related objects to make them eligible for garbage collection at the same time.

Purpose of using:

Reference : reUse the object if not yet GCed????

ReferentQueue: Do the clean-up of related objects.

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

Unlike soft and weak references, phantom references are not automatically cleared by the garbage collector as they are enqueued. An object that is reachable via phantom references will remain so until all such references are cleared or themselves become unreachable.

So is Phantom < Weak < Soft ?

**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.

**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.

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.

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

Calling Garbage collector: System.gc and Runtime.getRuntime.gc() are same.

System{

public static void gc() {

Runtime.getRuntime().gc();

}

}

**The permanent generation space (permgen) has been replaced with a new space called metaspace from Java 8**

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 (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.

Permanent generation stores the Meta data of the class. From Java 8 on wards the permanent generation is replaced by Meta Space which automatically grows in size (default behaviour)

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.

Both the young and old generation collections in Parallel GC are parallel and stop-the-world. Old generation collections also perform compaction.

### Concurrent Mark & Sweep (CMS) GC

**Young generation** :

In CMS GC, young garbage collections are like those of Parallel GC. They are parallel stop-the-world, meaning all Java application threads are paused during young garbage collections and the garbage collection work is performed by multiple threads.

**Old generation** :

The major difference between Parallel GC and CMS GC is the old generation collection.

1. It stops-the-world during the initial marking.
2. Does marking again in parallel with application threads.
3. STW for remarking again, all threads are suspended and all remaining newly allocated objects during step2 are marked as alive or unreachable.
4. Concurrent sweeping and compactions run in parallel with the application threads.

Chart, diagram

Description automatically generated with medium confidence

<https://www.informit.com/articles/article.aspx?p=2496621>

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.

### 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.

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

In the place of PermGen, a new feature called Meta Space has been introduced. MetaSpace grows automatically by default. Here, the garbage collection is automatically triggered when the class metadata usage reaches its maximum metaspace size.

When is GC triggered ? How does this GC trigger differ from the automatic GC of MetaSpace.

# 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.

## Bubble Sort

The main logic is inside the inner for loop.

1. The logic is , compare the current element with its next element.

If it is greater, then swap. So, by the end of first cycle the largest element has reached its final/correct position.

1. Since in every cycle the nth largest element will reach its position,

i.e In 1st cycle first largest element will reach its place

In 2nd cycle second largest element will reach its place

and so on

1. So we use an outer loop and make sure that the inner loop does not compare again with the already settled elements.

Improvement to this Algo : If there are no swaps during any cycle, it means the array is sorted and we can break out.

# Collection

<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.

public final int hashCode() {….}

public boolean equals(Object o) {

if o is equal to “this”, return true

if o is not a instance of thisClass return false

compare this.xyz with o.xyz

## Iterable

An interface which tells that the collection is Iterable. And to get an Iterator of that collection we use the methods Iterator().

The forEach() provides easy 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, ConcurrentHashMap.

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.

CopyOnWriteArrayList is used when the list is used more for iteration and less of modification. When we take an iterator, we get an immutable snapshot of the current list.

### ListIterator

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

## Collections Class

Provides lot of util methods.

## 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 .

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

A Priority Queue is different from a normal queue, because instead of being a “first-in-first-out”, values come out in order by priority. Uses MinHeap by default.

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

Actually, internal data structure of PriorityQueue is not ordered, it is a heap. The only guarantee provided by PriorityQueue is that poll(), peek(), etc return the least element.

#### 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.

Put in map :-

Map replaces & returns the old value(Key/Value) if the key is already present.

Map returns null if it’s a new key. Set (which internally uses map)checks for this null value and confirms.

**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**:

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.

In ConcurrentHashMap, the lock is applied to a segment instead of an entire Map. Each segment manages its own internal hash table. The lock is applied only for update operations. Collections.synchronizedMap(Map) synchronizes the entire 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: initial 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(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() returns zero for a null key, so it stores only one null key, that too 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 int hash;

final K key;

V value;

Entry<K,V> next; //to point to the next pair in of the same bucket.

}

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

**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)

#### TreeMap

Learn something 😉

#### 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 concurrentHashMap.

Hashtable ht = new Hashtable ();

ht.put(null, null);

#### concurrentHashMap

Provides Thread safety is ensured by having separate locks for separate buckets, resulting in better performance. Performance is further improved by providing read access concurrently without any blocking.

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

In ConcurrentHashMap, the lock is applied to a segment instead of an entire Map. Each segment manages its own internal hash table. The lock is applied only for update operations.

### Set

Internally uses Map.

Always inserts a dummy Object(new Object()) for the Value in key/value pair.

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

#### HashSet

Internally uses HashMap

1. Order: Ascending
2. Null : not allowed

#### LinkedHashSet

Internally uses LinkedHashMap

#### TreeSet

Internally uses TreeMap.

The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided.

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  Log(n) is almost close to log(1). |
| 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:**

This class implements the following patterns internally:

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 to perform an operation (or set of operations) in a particular manner.

When Strategy Pattern is implemented/used by the **aggregation** relation the class can be very must extendable.

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

Strategy pattern sometimes uses DI to switch between different strategy.

Collections.*sort*(list, **new** comparatorImplementor());//Sorting based on a statergy(passed as a comparatorImplementor)

### 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.

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/signature/contract of a class into another interface signature/contract 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 definition says, the subclass of the factory decides which class to instantiate. So, the logic should be in the subclass (The Concrete Class).

Factory pattern vs strategy pattern : 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

**Ex3:** Executor Class

**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, so we are localizing the object creation process.
* 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.

Instead of "new Object()" we are calling Factory.createNewObjet(parameters). How is it going to make difference?

There is or there must be some business logic that dynamically determines which object has to be created. This logic is localized in the factory pattern.

### 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 Abstract Factory**

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.***

**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 are possible because of multiple VM, multiple class Loader

**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.

public static Singleton getInstance3()

{

if (instance == null) {

synchronized (Singleton.class)

{

if (instance == null)

instance = new Singleton();

}

}

return instance;

}

**Eager Initialization vs Lazy Initialization**:

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 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.

Combo of Early and late Initialization and avoiding locking:

Notice the private inner static class that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the getInstance method, this class gets loaded and creates the Singleton class instance.

public class BillPughSingleton {

private BillPughSingleton(){}

private static class SingletonHelper{

private static final BillPughSingleton INSTANCE = new BillPughSingleton();

}

public static BillPughSingleton getInstance(){

return SingletonHelper.INSTANCE;

}

}

**volatile case**

asdsadsa

**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.

https://stackoverflow.com/questions/70689/what-is-an-efficient-way-to-implement-a-singleton-pattern-in-java

**Understand the Singleton scenarios when:**

**1.Reflection**.

Reflection can be disabled by :

Extend the SecurityManager class and override this method to restrict reflection access. If this method is overridden, then super.checkPackageAccess should be called as the first line in the overridden method.

public void checkPackageAccess(String pkg){

super.checkPackageAccess(pkg)

// don't allow the use of the reflection package

if(pkg.equals("java.lang.reflect")){

throw new SecurityException("Reflection is not allowed!");

}

}

If Reflection is **not** disabled, then :

We can throw exception because ,it is impossible that the static final instance is null after class is loaded. i.e. the static modifier of the “instance” variable will make sure that the singleton object is created during class loading. But if someone changed the modifier of constructor to public from private. And then invoked the constructor directly then if check will pass and throw the RuntimeException

class Reflection{

private **static** final Singleton instance = new Singleton();

**private** Singleton() {

if (instance != null)

throw new IllegalStateException("Creating second instance of this class.");

}

public static Singleton getInstance() { return instance; }

}

With Reflection we can break private modifier 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 deserialization

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. The object returned should be of a type that is compatible with all uses. If it is not compatible, a ClassCastException will be thrown when the type mismatch is discovered.

**4.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 ? – No, as enum already extends

**5.Class Loader**:

public class Foo{ public void FooMethod(){ Singleton.getInstance().doSomething(); } }

public class Boo{ public void BooMethod(){ Singleton.getInstance().doSomething(); } }

When class Foo and Boo use a different class loader, which means the singleton class is also loaded twice, resulting into two singleton instances.

### 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. It’s 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 implemented 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 member variable.

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 class**

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>

Inversion of Control - this is the principle of object-oriented programming, in which objects of the program do not depend on concrete implementations of other objects but may have knowledge about their abstractions (interfaces) for later interaction.

# 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).

**Thread States**:

* 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 processor(s).

**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.

Multithreading can be achieved by multitasking(Single processor) or by multiprocessing(more processors).

**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

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

## 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). But the presence of ExecutorService threads prevents the JVM from shutting down.

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 in sleep state until notify() or notifyAll() is called.

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

## Interrupt

Thread.interrupt() sets the interrupted status/flag of the target thread. Then code running in that target thread MAY poll the interrupted status and handle it appropriately. Some methods that block such as Object.wait(), join,sleep may consume the interrupted status immediately and throw an appropriate exception (usually InterruptedException)

The target thread is "interrupted". Mostly, a flag is set in that thread, which the thread can look at (with Thread. isInterrupted ()). If the target thread was currently blocked on some I/O or Object.wait(), then it is awakened with, respectively, an InterruptedIOException or an InterruptedException.

Thread interruption is a gentle way to nudge(other than wait, join, sleep) a thread. It is used to give threads a chance to exit cleanly, as opposed to Thread.stop(), which is more like shooting the thread with an assault rifle.

## 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 or written from or to the main memory. The intermediate CPU cache or registers are not used.

**DrawBacks**:

* Because of skipping the cache and registers the use of volatile is time consuming.
* Use of volatile variables also prevent instruction reordering which is a normal performance enhancement technique.

Volatile variable may be helpful when doing operation which are atomic.32 bit (long and double)

When a volatile instance variable is read or written from or to the main memory other instance variables of that class is also read or written. Otherwise, the other instance variable in the main memory would be stale/inconsistent.

### Happens-Before Guarantee

Few points to understand : (Visible-Volatile-Guarantee-ReOrdering)

1. Visibility problem : between thread because of CPU cache and registers.
2. So Volatile is used which solved the visibility problem.
3. Volatile Guarantee : every read/write of volatile variable causes read/write of the other variable of the instance. Otherwise, inconsistent object will be the main memory.
4. Instruction reordering problem: Because of the volatile’s guarantee, the other variable’s wrong data is getting updated into the main memory.

Example Problem : Change date from : 28 Feb 2022 to : 31 Mar 2022

Class BirthDay {

Int volatile days

Int months

Int years

update(){

this.days= days

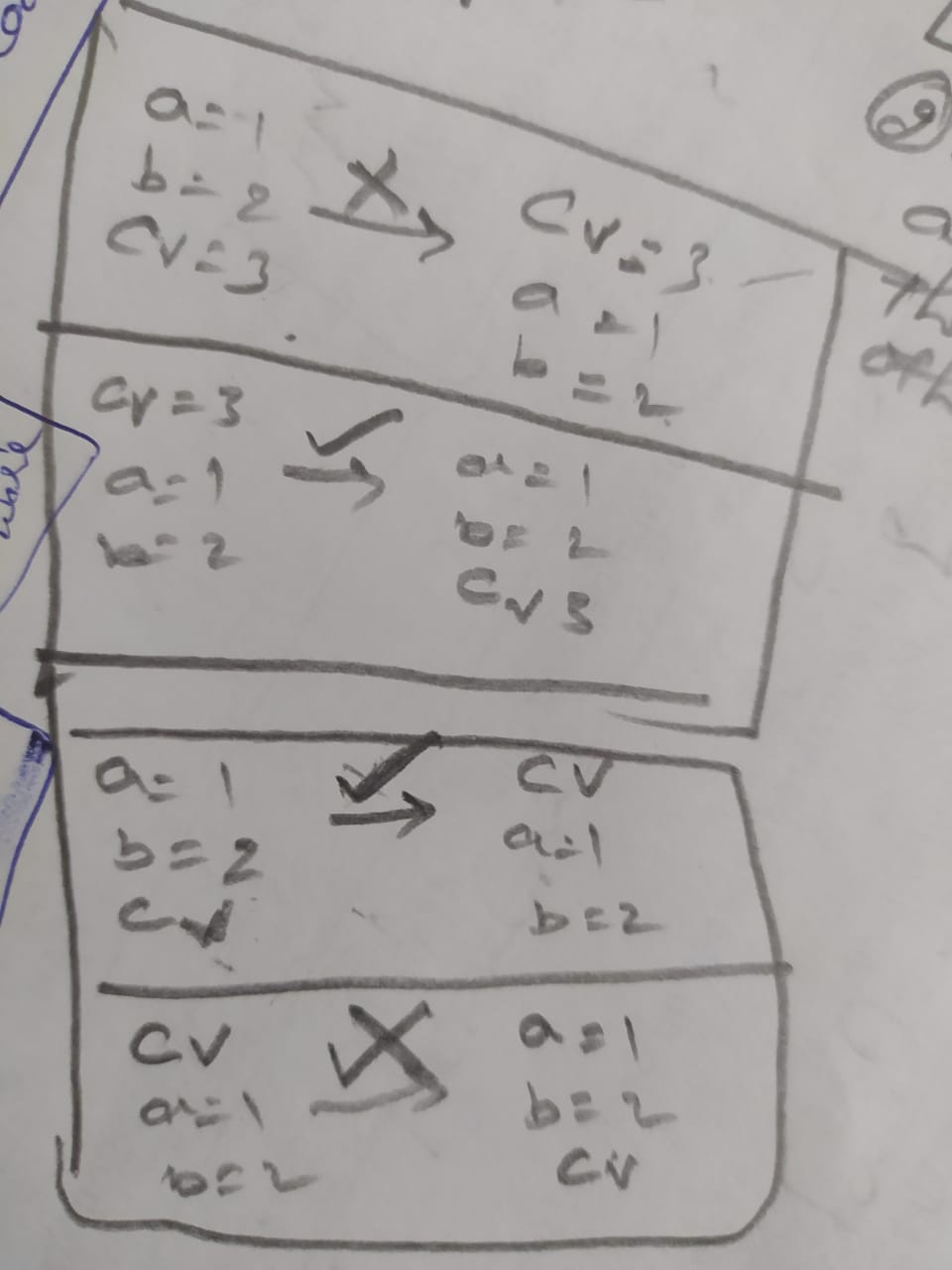
this.months=months

this.years =years

}

}

Hence, the volatile provides this “Happens-Before” guarantee, which is :



Happens-before defines a partial ordering on all actions within the program. In the absence of a happens-before, JVM is free to reorder the instruction for better performance.

Happens-before is not just reordering of actions in 'time' but also a guarantee of ordering of read and write to memory .

Happens-before is a concept, a phenomenon, or simply a set of rules that define the basis for reordering of instructions.it is simply a discipline put into place so that in a multi-threading environment the reordering of the surrounding instructions does not result in a code that produces incorrect output.

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.

## Intrinsic Lock or Monitor Lock

Synchronization is built around an internal entity known as the intrinsic lock or monitor lock. Every object has an intrinsic lock associated with it. When a thread owns an intrinsic lock of an Object, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock. when a static synchronized method is invoked, the thread acquires the intrinsic lock for the Class object associated with the class.

## 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 keyword : methods and blocks.
* Locks : methods, blocks, segment/snippets.
* Atomic variables like Atomic Integer.
* Mutex concept.
* 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, synchronized(this){….}
* 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. Yes

DIFFERENCE between Synchronization and volatile , in terms of reading from main memory.

**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? No.

// Custom implementation of ReEntrantLock

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 returns 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.

## 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 semantics 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.

There is only 2 classes : ReentrantLock and ReentrantReadWriteLock

### 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 Object. The access 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. Write lock is exclusive.

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

## Callable, FutureTask

* Callable has the extra ability to return an Object.
* It can also throw an Exception.

The exception thrown by the call() can be retrieved from the Future.get(). That is, in the catch clause of the Future.get() we will get the exception.

Callable on its own will not do the job like the Runnable. We need a Future object to wrap Callable and return to a value or exception.

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

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.

Useful methods of FutureTask :

* get() : a blocking call which waits until the call() of Callable returns the generic type.
* get(timeout, TimeUnit), waits until the specified time and the returns.
* cancel(mayInterruptIfRunning) :
* isDone() :

**The FutureTask run() is calling the call() of Callable.**

Whichever exception is thrown by the Callable of the FutureTask, we receive the exception as ExecutionException. Actual exception can be retrieved by e.getCause();

## Queues 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.

All BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control.

Unlike the regular collection (arraylist,..) the size of the blocking queue can be bounded by specifying the size in the constructor while creating the queue.

Note, Synchronous queue does not have a constructor to specify the size. It has only a Boolean constructor to give fairness to the waiting threads.

#### 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

## 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.

<https://jenkov.com/tutorials/java-concurrency/thread-pools.html> . A custom implementation of 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 ExecutorService and execute the tasks(Runnable)

Diagram

Description automatically generated

### 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 Interface

Besides the execute() method, this interface also defines a similar submit() method that can return a Future object, Which is the main reason for choosing submit method instead of execute method.

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 for all tasks to be completed. If that time expires, the execution is stopped immediately.

**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().

**Rejection Handlers & Polices(While shutdown of executor service):**

Discard: Silently discard without any response.

Discard oldest: Discard the oldest.

Abort policy: Rejected Execution Exception.

Caller run policy:

##### 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);

We can schedule tasks to run after a given delay, or to execute periodically.

Furthermore, the interface defines two additional methods:

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

scheduleWithFixedDelay() : Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one execution and the commencement of the next. If any execution of the task encounters an exception, subsequent executions are suppressed. Otherwise, the task will only terminate via cancellation or termination of the executor.

scheduleAtFixedRate( () -> print("Fixed Rate Scheduled"), initialDelay, period, TimeUnit.MILLISECONDS);

scheduleAtFixedRate() :starts executing the task after 2 ms delay, then repeats it at every 2 seconds.

executions will commence after initialDelay then initialDelay+period, then initialDelay + 2 \* period, and so on. If any execution of the task encounters an exception, subsequent executions are suppressed. Otherwise, the task will only terminate via cancellation or termination of the executor. If any execution of this task takes longer than its period, then subsequent executions may start late, but will not concurrently execute.

<https://jenkov.com/tutorials/java-util-concurrent/scheduledexecutorservice.html#scheduleatfixedrate>

Rate : It’s about starting.

Delay : It’s about ending .

##### ThreadPoolExecutor

All the Executors internally uses ThreadPoolExecutor class with default configuration as you can see above. Now there are scenarios where default configuration is not suitable say instead of LinkedBlockingQueue a priority queue needs to be used etc. In such cases caller can directly work on underlying ThreadPoolExecutor by instantiating it and passing desired configuration to it.

new ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,

long keepAliveTime, TimeUnit unit,

BlockingQueue<Runnable> workQueue);

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.

##### Executors class Factory

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.

1. **SingleThreadExecutor** :

Tasks are placed in a LinkedBlockingQueue.

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

1. **FixedThreadPool**

Creates specified number of threads. If the tasks are more than the available threads, they are stored in LinkedBlockingQueue.

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

1. **CachedThreadPool**

Every time a task is created it is posted into the Synchronous Queue(Blocking queue,single storage area, ). If no threads are free, it will create a new thread and assign the task. If any thread is ideal for more than a specific time, it will be killed.

ExecutorService executorService = Executors.newCachedThreadExecutor();

The queue size will always be zero.

1. **ScheduledExecutor**

DelayedWorkQueue.

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

For Fixed thread pool and single thread pool uses Linked blocking queue data structure because they maintain a fixed number of threads. Since the threads are limited, the task will pile up and hence it uses linked blocking queue which can be expanded easily.

Cached thread pool maintains unlimited thread to maintain zero threads in waiting list, so it uses a synchronous queue which is also a blocking queue.

Apart form of main single task queue, each thread in the ForkJoinPool has its own double-ended queue which stores its subtasks.

What are the factors that influence us in choosing any one of the above factory methods.

##### ForkJoinPool

ForkJoinPool forkJoinPool = new ForkJoinPool(4);

This example creates a ForkJoinPool with a parallelism level of 4.

**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.

## 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 thread will 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 Reused.

### Semaphores

The permit machines.

There are "binary" semaphores and "counting/general" semaphores. Java's semaphore is a counting semaphore and thus allows it to be initialized with a value greater than zero.

A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore) and can signal that they have finished using the resource (incrementing the semaphore).

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.

acquire() : acquires a permit and blocking until one is available.

tryAcquire() : return true if a permit is available immediately and acquire it otherwise return false.

release() : release a permit.

availablePermits() : return number of current permits available.

### Mutex

Mutex is typically used to serialize access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.

There are various ways, we can implement a mutex in Java:

1. Using synchronized Keyword : The synchronized method/block are locked using a instance varible object which would be same for all the threads of a instance.

2. Using ReentrantLock

3. Using Semaphore

**Mutex vs Semaphrone**:

1. Mutex gives only one permit.

2. Mutex is block wise. Semaphrone can acquire and release from different places.

3.

Is one permit semaphore equals to mutex ?

### 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.

It can also be used to synchronize a single phase, and in that regard, it acts much like a CyclicBarrier.