# Java : Introduction

### 4 Stages of Becoming a Programmer

1. **Unconscious Incompetence:** Decisions are hard to make.
2. **Conscious Incompetence:** Recognizing the skills needed.
3. **Conscious Competence:** Training is now starting to make sense.
4. **Unconscious Competence:** Ultimate level - where we want to be, programming become second nature.

### Tips for Self-Taught Software Developers

1. Best Practices
2. Create a project and put it on GitHub: Something to show to prospective employers.
3. Study other people code

## About Java

1. **Java is a simple language.**
    - Java is easy to learn and its syntax is clear and concise.
    - It is based on C++ (so it is easier for programmers who know C++).
    - Java has removed many confusing and rarely-used features e.g. explicit pointers, operator overloading etc.
    - Java also takes care of memory management and it also provides an automatic garbage collector.
    - This collects the unused objects automatically.
2. **Java is a platform-independent language.**
    - Java programs after compilation, are converted into an intermediate level language called the **bytecode**.
    - **bytecode** is a part of the Java platform irrespective of the machine on which the programs run.
    - This makes java highly portable as its bytecodes can be run on any machine by an interpreter called the **JVM**.
    - Thus java provides **'reusability of code'**.
3. **Java is an object-oriented (OOP) programming language.**
    - OOP makes the complete program simpler by dividing it into a number of objects.
    - The objects can be used as a bridge to have data flow from one function to another.
    - We can easily modify data and function's as per the requirements of the program.
4. **Java is a robust language.**
    - Java programs must be reliable because they are used in both consumer and mission-critical applications.
    - Ranging from Blu-ray players to navigation systems.
5. **Java is a multithreaded language.**
    - Java can perform many tasks at once by defining multipe threads.
    - ***Example:*** A program that manages a Graphical User Interface (GUI) while waiting for input from a network connection uses another thread to perform and wait's instead of using the default GUI thread for both tasks. This keeps the GUI responsive.
6. **Java programs can create applets.**
    - Applets are programs that run in web browsers.
7. **Java does not require any preprocessor.**
    - It does not require inclusion of header files for creating a Java application.

## Naming Conventions

### 1. Packages:

- The prefix of a unique package name is always written in **all-lowercase ASCII letters** and should be one of the top-level domain names, like com, edu, gov, mil, net, org.
- Subsequent components of the package name vary according to an organisation's own internal naming conventions.

```
com.sun.eng
com.apple.quicktime.v2

// java.lang packet in JDK
java.lang
```

### 2. Classes and Interfaces:

- Class names should be **nouns**, in mixed case with the **first** letter of each internal word capitalised. Interfaces name should also be capitalised just like class names.

- Use whole words and must avoid acronyms and abbreviations.

```
interface  Bicycle
class MountainBike implements Bicyle

interface Sport
class Football implements Sport
```

## 3. Methods:

- Methods should be **verbs**, in mixed case with the **first letter lowercase** and with the first letter of each internal word capitalised.

```
void changeGear(int newValue);
void speedUp(int increment);
void applyBrakes(int decrement);
```

## 4. Variables:

- Variable names should be short yet meaningful.
- Should **not** start with underscore('_') or dollar sign '$' characters.
- Should be mnemonic i.e, designed to indicate to the casual observer the intent of its use.
- **One-character variable names should be avoided** except for temporary variables.
- Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

```
// variables for MountainBike class
int speed = 0;
int gear = 1;
```

## 5. Constant variables:

- Should be **all uppercase** with words separated by underscores ("_").
- There are various constants used in predefined classes like Float, Long, String etc.

```
static final int MIN_WIDTH = 4;

// Some  Constant variables used in predefined Float class
public static final float POSITIVE_INFINITY = 1.0f / 0.0f;
public static final float NEGATIVE_INFINITY = -1.0f / 0.0f;
public static final float NaN = 0.0f / 0.0f;
```

# Handling Java Installation and Un-installation

This article will tell you how to install / uninstall multiple java versions on mac os both use home brew or manually. It also show you how to set **JAVA_HOME** & **PATH** system environment variable to make your installed jdk as the default jdk. We recommend you to use home brew, because it provide a lot of easy to use tools.

## 1. Use Home Brew To Install & Uninstall Java On Mac OS X.

### 1.1 Use Brew Cask Command To Install Java JDK.

1. Install HomeBrew on your mac os, please read article How To Install Homebrew On Mac OS.
2. Run below command to install home brew cask extension, cask extension tool can help you to install and manage software package easily.

```
3.  $ brew tap homebrew/cask-versions
4.  Updating Homebrew...
5.  ==> Auto-updated Homebrew!
6.  Updated 1 tap (homebrew/core).
7.  ==> Updated Formulae
8.  node-build
9.  ==> Tapping homebrew/cask-versions
10. Cloning into '/usr/local/Homebrew/Library/Taps/homebrew/homebrew-cask-versions'...
11. remote: Enumerating objects: 198, done.
12. remote: Counting objects: 100% (198/198), done.
13. remote: Compressing objects: 100% (194/194), done.
14. remote: Total 198 (delta 9), reused 29 (delta 1), pack-reused 0
15. Receiving objects: 100% (198/198), 84.76 KiB | 221.00 KiB/s, done.
16. Resolving deltas: 100% (9/9), done.
17. Tapped 169 casks (216 files, 324.9KB).
```

18. Run `brew search java` or `brew search jdk` command to find your desired java or jdk version.
    Below is search result for `brew search java` command.

```
 $ brew search java
 ==> Formulae
 app-engine-java                                google-java-format                                javarepl
jslint4java                                    libreadline-java
 ==> Casks
 charles-applejava      eclipse-java           eclipse-javascript     java ✔                 java-beta
java11                 java6                  netbeans-java-ee       netbeans-java-se       oracle-jdk-javadoc
yourkit-java-profiler
```

```
 charles-applejava      eclipse-java           eclipse-javascript     java ✔                 java-beta
java11                 java6                  netbeans-java-ee       netbeans-java-se       oracle-jdk-javadoc
yourkit-java-profiler
```

Below is search result for **brew search jdk** command.

```
 $ brew search jdk
 ==> Casks
 adoptopenjdk              adoptopenjdk              adoptopenjdk8             adoptopenjdk8             oracle-
jdk                oracle-jdk                oracle-jdk-javadoc        oracle-jdk-javadoc        sapmachine-jdk
sapmachine-jdk
```

19. Now install the jdk version that you need like below. During the installation process, you may encounter some error message like

    **Error: Cask 'java8' is unavailable: No Cask with this name exists** or **Error: Cask adoptopenjdk8 exists in multiple taps.**

```
 $ brew cask install java8
 Error: Cask 'java8' is unavailable: No Cask with this name exists.

 $ brew cask install adoptopenjdk8
 Error: Cask adoptopenjdk8 exists in multiple taps:
   homebrew/cask-versions/adoptopenjdk8
   caskroom/versions/adoptopenjdk8
```

20. **Below is the correct command to install jdk 8 use home brew cask command**

    `$ brew cask install homebrew/cask-versions/adoptopenjdk8`

    You can also run `$ brew cask install java` to install the newest jdk version.

    In this example, i had installed the newest jdk version 12.0.1.

```
 $ brew cask install homebrew/cask-versions/adoptopenjdk8
 ==> Satisfying dependencies
 ==> Downloading https://github.com/AdoptOpenJDK/openjdk8-binaries/releases/download/jdk8u212-b03/OpenJDK8U-
jdk_x64_mac_hotspot_8u212b03.pkg
 ==> Downloading from https://github-production-release-asset-2e65be.s3.amazonaws.com/140418865/07e4b900-61d1-
11e9-96f2-868c40733c49?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20190603%2Fus-
east-1%2Fs3%2Faws4_request&X-Amz-Date=2019
 ######################################################################## 100.0%
 ==> Verifying SHA-256 checksum for Cask 'adoptopenjdk8'.
 ==> Installing Cask adoptopenjdk8
 ==> Running installer for adoptopenjdk8; your password may be necessary.
 ==> Package installers may write to any location; options such as --appdir are ignored.
 Password:
 installer: Package name is AdoptOpenJDK
 installer: Installing at base path /
 installer: The install was successful.
 🍺  adoptopenjdk8 was successfully installed!
```

21. After install you can first run `/usr/libexec/java_home` command to get the java home directory, then go to the jdk home directory to see all the installed jdk.

```
# Get the java home directory info.
$ /usr/libexec/java_home
/Library/Java/JavaVirtualMachines/jdk-12.0.1.jdk/Contents/Home
# List all the installed jdk in java home directory.
$ ls -l /Library/Java/JavaVirtualMachines/
total 0
drwxr-xr-x  3 root      wheel  96 Jun  3 11:19 adoptopenjdk-8.jdk
drwxr-xr-x  3 root      wheel  96 May  3 22:31 jdk-12.0.1.jdk
drwxr-xr-x@ 3 songzhao  staff  96 Apr  2 16:23 openjdk-12.0.1.jdk
```

31. From above output we can see that there are three java jdk that has been installed, two open jdk and one standard jdk. Run each jdk's java executable command, you can get below output.

```
$ ./openjdk-12.0.1.jdk/Contents/Home/bin/java -version
openjdk version "12.0.1" 2019-04-16
OpenJDK Runtime Environment (build 12.0.1+12)
OpenJDK 64-Bit Server VM (build 12.0.1+12, mixed mode, sharing)
$ ./adoptopenjdk-8.jdk/Contents/Home/bin/java -version
openjdk version "1.8.0_212"
OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_212-b03)
OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.212-b03, mixed mode)
$ ./jdk-12.0.1.jdk/Contents/Home/bin/java -version
java version "12.0.1" 2019-04-16
Java(TM) SE Runtime Environment (build 12.0.1+12)
Java HotSpot(TM) 64-Bit Server VM (build 12.0.1+12, mixed mode, sharing)
```

## 1.2 Use Brew Cask Command To UnInstall JDK.

1. Run home brew cask command `$ brew cask uninstall java` to uninstall open jdk 12.0.1.

   If you want to uninstall open jdk 8 then run `$ brew cask uninstall caskroom/versions/adoptopenjdk8`

   **Below command uninstall open jdk 12.0.1.**

```
 $ brew cask uninstall java
 ==> Uninstalling Cask java
 ==> Backing Generic Artifact 'openjdk-12.0.1.jdk' up to
'/usr/local/Caskroom/java/12.0.1,69cfe15208a647278a19ef0990eea691/jdk-12.0.1.jdk'.
 Password:
 ==> Removing Generic Artifact '/Library/Java/JavaVirtualMachines/openjdk-12.0.1.jdk'.
 ==> Removing directories if empty:
 /Library/Java/JavaVirtualMachines
 ==> Purging files for version 12.0.1,69cfe15208a647278a19ef0990eea691 of Cask java
```

   **Below command uninstall open jdk 8.**

```
 $ brew cask uninstall caskroom/versions/adoptopenjdk8
 ==> Uninstalling Cask adoptopenjdk8
 ==> Uninstalling packages:
 net.adoptopenjdk.8.jdk
 Password:
 ==> Purging files for version 8,212:b03 of Cask adoptopenjdk8
```

During above jdk uninstall process, you may encounter below error, **Error: Cask adoptopenjdk8 exists in multiple taps**.

This is because you should uninstall **caskroom/versions/adoptopenjdk8** instead of **adoptopenjdk8**.

```
$ brew cask uninstall adoptopenjdk8
Error: Cask adoptopenjdk8 exists in multiple taps:
  homebrew/cask-versions/adoptopenjdk8
  caskroom/versions/adoptopenjdk8
```

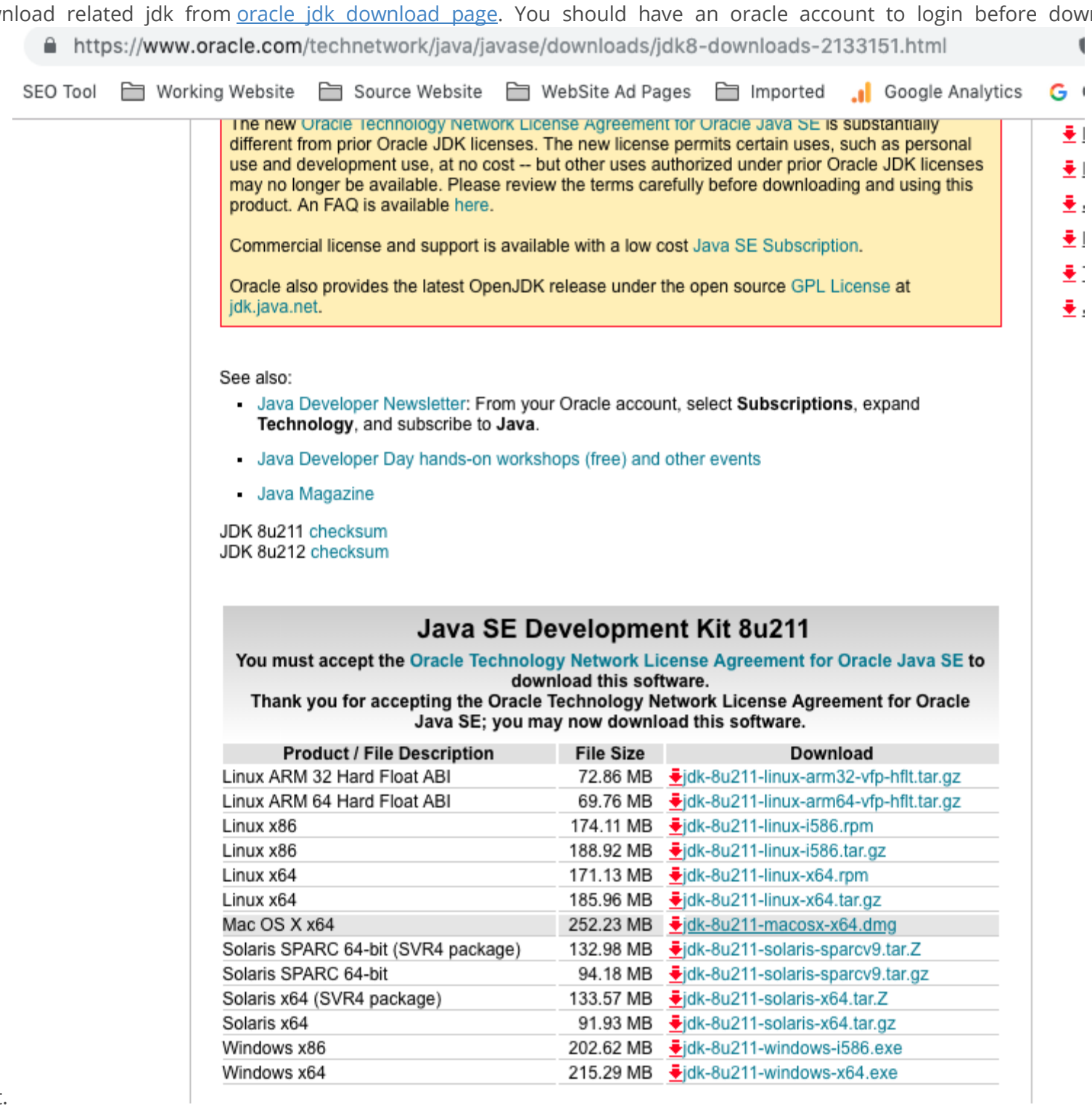
2. Now go to jdk installation directory, you will find above two home brew installed jdk directory has been removed.

```
$ ls -l /Library/Java/JavaVirtualMachines/
total 0
drwxr-xr-x  3 root  wheel  96 May  3 22:31 jdk-12.0.1.jdk
```

# 2. Install & Uninstall JDK In Mac OS Manually.

## 2.1 Install JDK Manually In Mac OS X.

1. Download related jdk from oracle jdk download page. You should have an oracle account to login before download start.




2. Double click the downloaded dmg file to install jdk, just follow the wizard to install it.
3. When the installation complete, you can find the jdk install directory in folder **/Library/Java/JavaVirtualMachines/**.

   From below output, we can see there are three jdk has been installed, one ( adoptopenjdk-8.jdk ) is installed with home brew, the other two ( jdk-12.0.1.jdk, jdk1.8.0_211.jdk ) are installed by download installation file from oracle.

```
$ ls -l /Library/Java/JavaVirtualMachines/
total 0
drwxr-xr-x  3 root  wheel  96 Jun  3 14:02 adoptopenjdk-8.jdk
drwxr-xr-x  3 root  wheel  96 May  3 22:31 jdk-12.0.1.jdk
drwxr-xr-x  3 root  wheel  96 Jun  3 15:54 jdk1.8.0_211.jdk
```

## 2.2 Uninstall Java JDK Manually In Mac OS X.

Follow oracle's documents, to manually uninstall installed jdk in mac os x, you just need to remove some files with root permission in terminal like below.

1. Open a terminal and run below command to remove java jdk installed directory.

```
$ sudo rm -rf /Library/Java/JavaVirtualMachines/jdk1.8.0_211.jdk

$ ls -l /Library/Java/JavaVirtualMachines/
total 0
drwxr-xr-x  3 root  wheel  96 Jun  3 14:02 adoptopenjdk-8.jdk
drwxr-xr-x  3 root  wheel  96 May  3 22:31 jdk-12.0.1.jdk
```

7. Run below command to remove java control panel if exist.

```
$ sudo rm -rf /Library/PreferencePanes/JavaControlPanel.prefPane
```

9. Remove java applet plugins.

```
$ sudo rm -rf /Library/Internet\ Plug-Ins/JavaAppletPlugin.plugin/
```

11. Remove java application support.

```
$ sudo rm -rf ~/Library/Application\ Support/Java
```

13. Remove java updater list.

```
$ sudo rm -rf /Library/LaunchAgents/com.oracle.java.Java-Updater.plist
$ sudo rm -rf /Library/PrivilegedHelperTools/com.oracle.java.JavaUpdateHelper
```

16. Remove java helper tool.

```
$ sudo rm -rf /Library/LaunchDaemons/com.oracle.java.Helper-Tool.plist

$ sudo rm -rf /Library/Preferences/com.oracle.java.Helper-Tool.plist
```

## 3. Set JAVA_HOME & PATH System Environment Variable.

Now we have known how to install / uninstall multiple java jdk versions on mac os. But after that you need to set **JAVA_HOME** & **PATH** environment variable value to set your installed jdk as default jdk. So that you can use it easily.

1. Go to user home directory.
2. Run `$ nano .bash_profile` command to open **.bash_profile** file to edit.
3. Insert below export command in **.bash_profile** file.

```
# Set JAVA_HOME system environment variable value.
export JAVA_HOME=/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home

# Add java bin folder in PATH system environment variable value.
export PATH=$PATH:$JAVA_HOME/bin
```

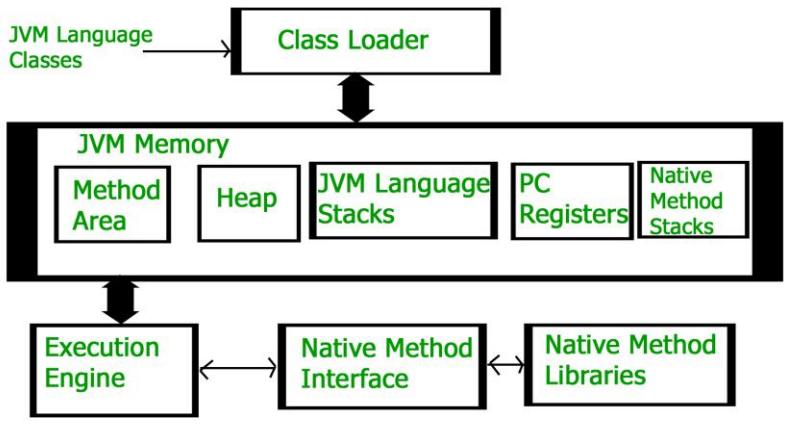
9. Save the above changes and quit editor.
10. Restart terminal and use command `$ java -version` to see the new jdk version is used by output info.

```
$ java -version
openjdk version "1.8.0_212"
OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_212-b03)
OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.212-b03, mixed mode)
```

# JVM Architecture : Working of JVM

### What is Java Virtual Machine (JVM) ?

- It acts as a run-time engine to run Java applications.
- It is the one that actually calls the **main** method present in a java code.
- JVM is a part of **JRE**(Java Runtime Environment).
- Java applications are called **WORA** (Write Once Run Anywhere).
- We can develop Java code on one system and run on any other Java enabled system without any adjustment. This is all possible because of JVM.
- When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.




### The 3 major section that the compiled program goes through when it is run are:

- Class Loader Subsystem
- JVM Memory
- Execution Engine

## Class Loader Subsystem

It is mainly responsible for three activities.

1. Loading
2. Linking
3. Initialization

## 1. Loading

- The Class loader **reads the *.class* file**, generate the corresponding binary data and save it in method area.
- For each ***.class*** file, JVM stores following information in method area.
    - Fully qualified name of the loaded class and its immediate parent class.
    - Whether ***.class*** file is related to Class or Interface or Enum.
    - Modifier, Variables and Method information etc.
- **After loading *.class* file**, JVM creates an object of type Class to represent this file in the heap memory.
- Please note that this object is of type Class predefined in ***java.lang package***.
- This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc.
- To get this object reference we can use *getClass()* method of **Object class** discussed in other important concepts section.

```
package com.learn.java.sec1.jvm_architecture;

import java.lang.reflect.Field;
import java.lang.reflect.Method;

class Student  {
    private String name;
    private int roll_No;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getRoll_no()  {
        return roll_No;
    }

    public void setRoll_no(int roll_no) {
        this.roll_No = roll_no;
    }
}


public class ClassObjectJVM {
    public static void main (String[] args) {
        Student s1 = new Student();

        // Getting hold of Class object created by JVM.
        Class c1 = s1.getClass();

        // Printing type of object using c1.
        System.out.println("Name of class: " + c1.getName());

        // getting all methods in an array
        System.out.println("\nAll the Declared Methods of the class: ");
        Method m[] = c1.getDeclaredMethods();
        for (Method method : m) {
            System.out.println(method.getName());
        }

        // getting all fields in an array
        System.out.println("\nAll the Declared Fields of the class: ");
        Field f[] = c1.getDeclaredFields();
        for (Field field : f) {
            System.out.println(field.getName());
        }

    }
}
```

**Output:**


Note:- For every loaded *.class* file, only one object of Class is created.

```
Student s2 = new Student();
```

```java
// c2 will point to same object where c1 is pointing
Class c2 = s2.getClass();
System.out.println(c1==c2); // true
```

## 2. Linking

- It Performs verification, preparation and (optionally) resolution.
- **Verification:**
  - It ensures the ***correctness of .class file*** i.e. it check whether this file is properly formatted and generated by valid compiler or not.
  - If ***verification fails***, we get run-time exception ***java.lang.VerifyError***.
- **Preparation:**
  - JVM ***allocates memory for class variables*** and initializing the memory to default values.
- **Resolution:**
  - It is the process of ***replacing symbolic references from the type with direct references***.
  - It is done by searching into method area to locate the referenced entity.

## 3. Initialization

- In this phase, all static variables are assigned with their values defined in the code and static block(if any).
- This is executed from top to bottom in a class and from parent to child in class hierarchy.
- In general, there are three class loaders :
  - **Bootstrap class loader:**
    - Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes.
    - It loads core java API classes present in ***JAVA_HOME/jre/lib*** directory.
    - This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
  - **Extension class loader:**
    - It is ***child of bootstrap class loader***.
    - It loads the classes present in the extensions directories ***JAVA_HOME/jre/lib/ext*****(Extension path)** or any other directory specified by the java.ext.dirs system property.
    - It is implemented in java by the ***sun.misc.Launcher$ExtClassLoader*** class.
  - **System/Application class loader:**
    - It is ***child of extension class loader***.
    - It is responsible to load classes from application class path.
    - It internally uses Environment Variable which mapped to java.class.path.
    - It is also implemented in Java by the ***sun.misc.Launcher$AppClassLoader*** class.

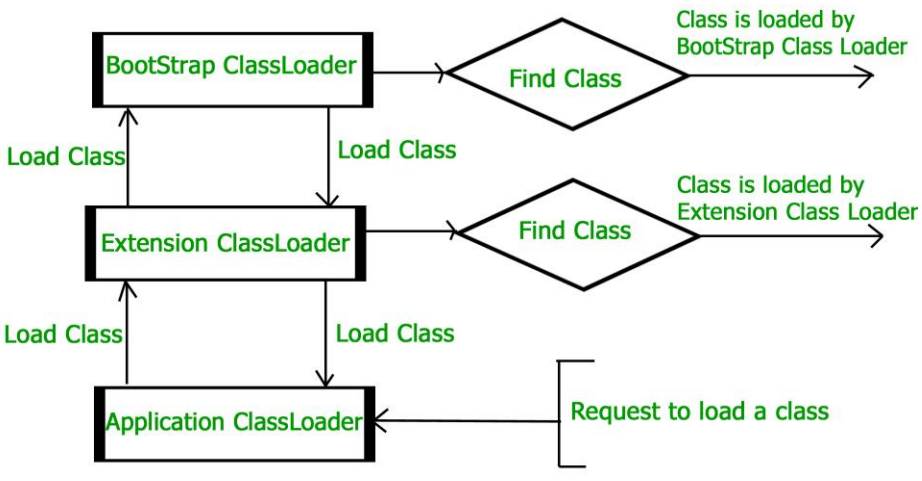
```java
public class Test {
    public static void main(String[] args) {
        // String class is loaded by bootstrap loader, and
        // bootstrap loader is not Java object, hence null
        System.out.println(String.class.getClassLoader());

        // Test class is loaded by Application loader
        System.out.println(Test.class.getClassLoader());
    }
}
```

**Output:**

```
null
sun.misc.Launcher$AppClassLoader@73d16e93
```

Notes:

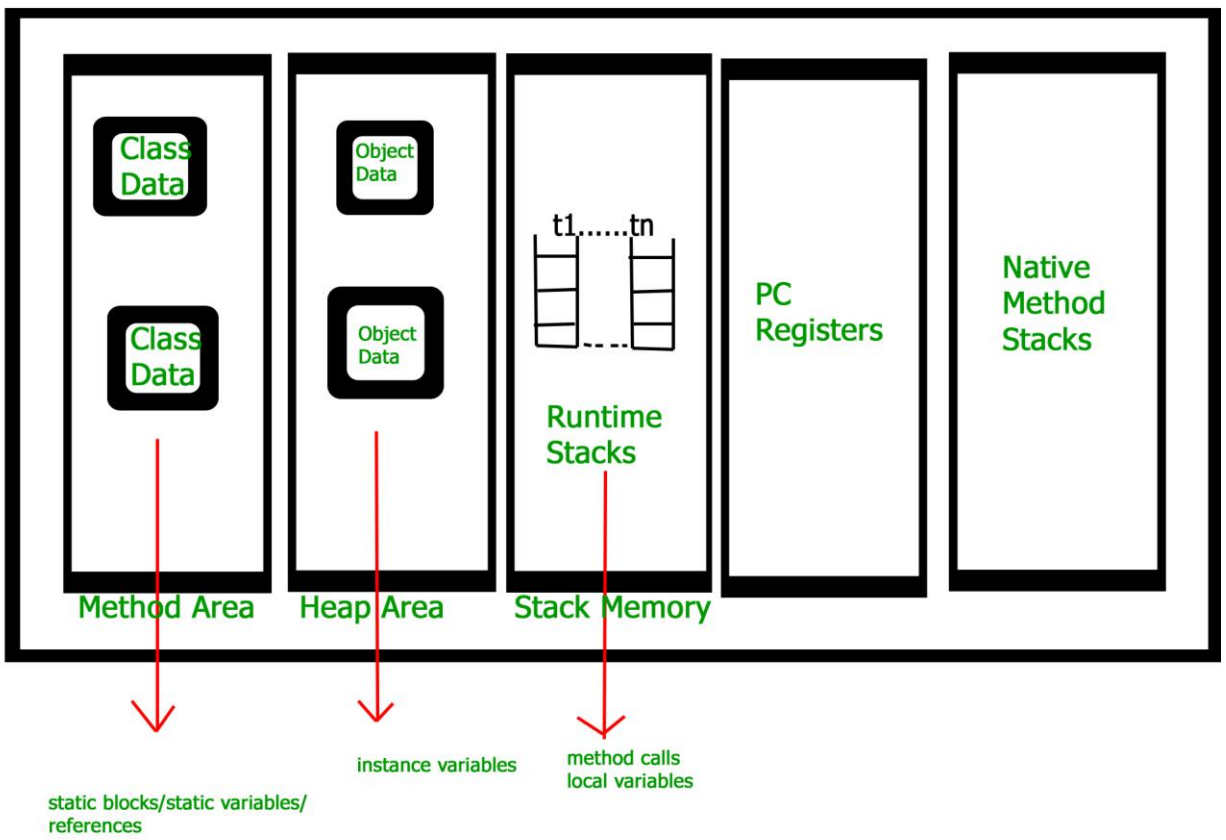
- JVM follow Delegation-Hierarchy principle to load classes.
- System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader.
- If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader.
- At last if system class loader fails to load class, then we get run-time exception ***java.lang.ClassNotFoundException***.



# JVM Memory

## Components of JVM Memory

- **Method area :**
  - In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables.
  - There is only one method area per JVM, and it is a shared resource.
- **Heap area :**
  - Information of all objects is stored in heap area.
  - There is also one Heap Area per JVM. It is also a shared resource.
- **Stack area :**
  - For every thread, JVM create one run-time stack which is stored here.
  - Every block of this stack is called activation record/stack frame which store methods calls.
  - All local variables of that method are stored in their corresponding frame.
  - After a thread terminate, it's run-time stack will be destroyed by JVM.
  - It is not a shared resource.
- **PC Registers :**
  - Store address of current execution instruction of a thread.
  - Obviously each thread has separate PC Registers.
- **Native method stacks :**
  - For every thread, separate native stack is created.
  - It stores native method information.




##Execution Engine

- Execution engine execute the *.class* (bytecode).
- It reads the byte-code line by line, use data and information present in various memory area and execute instructions.
- It can be classified in 3 parts:
  - **Interpreter:**
    - It interprets the bytecode line by line and then executes.
    - The disadvantage here is that when one method is called multiple times, every time interpretation is required.
  - **Just-In-Time Compiler(JIT):**
    - It is used to increase efficiency of interpreter.
    - It compiles the entire bytecode and changes it to native code.
    - So whenever interpreter see repeated method calls, JIT provide direct native code for that part so re-interpretation is not required, thus efficiency is improved.
  - **Garbage Collector:**
    - It destroy un-referenced objects.
    - Refer Garbage Collector.

### Java Native Interface (JNI).

- It is an interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution.

- It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.
- Native Method Libraries:
    - It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

---

# Java Basics

- Java is one of the most popular and widely used programming language and platform.
- A platform is an environment that helps to develop and run programs written in any programming language.
- Java is fast, reliable and secure.
- Java is used in every nook and corner from desktop to web applications, scientific supercomputers to gaming consoles, cell phones to the Internet.

## Java Environment

**JDK (Java Development Kit):**

- Intended for software developers.
- Includes development tools such as the Java compiler, Javadoc, Jar and a debugger.

**JRE (Java Runtime Environment):**

- JRE contains the parts of the Java libraries required to run Java programs.
- JRE can be view as a subset of JDK.
- Intended for end users.

**JVM (JVM (Java Virtual Machine):**

- It is an abstract machine.
- It is a specification that provides runtime environment in which java bytecode can be executed.
- JVMs are available for many hardware and software platforms.

## Java Basic Syntax

Simplest HelloWorld Program.

```
class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello, World");
    }
}
```

### 1. Class definition:

- Below line uses the keyword class to declare that a new class is being defined.

```
class HelloWorld
```

- HelloWorld is an identifier that is the name of the class.
- The entire class definition, including all of its members, will be between the braces {} .

### 2. main() method:

- In Java programming language, every application must contain a `main` method whose signature is:

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

- **public:** So that JVM can execute the method from anywhere.
- **static:** Main method is to be called without object.
- The modifiers public and static can be written in either order.
- **void:** The main method doesn't return anything.
- **main():** Name configured in the JVM.
- **String[]:** The main method accepts a single argument: an array of elements of type String.

**Note:** Like in C/C++, main method is the entry point of application and will subsequently invoke all the other methods required by program.

### 3. println() method:

- This line outputs the string “Hello, World” followed by a new line on the screen.

```
System.out.println("Hello, World");
```

- Output is actually accomplished by the ***built-in println( )*** method.
- **System** is a predefined class that provides access to the system.
- **out** is the variable of type output stream that is connected to the console.

Notes:

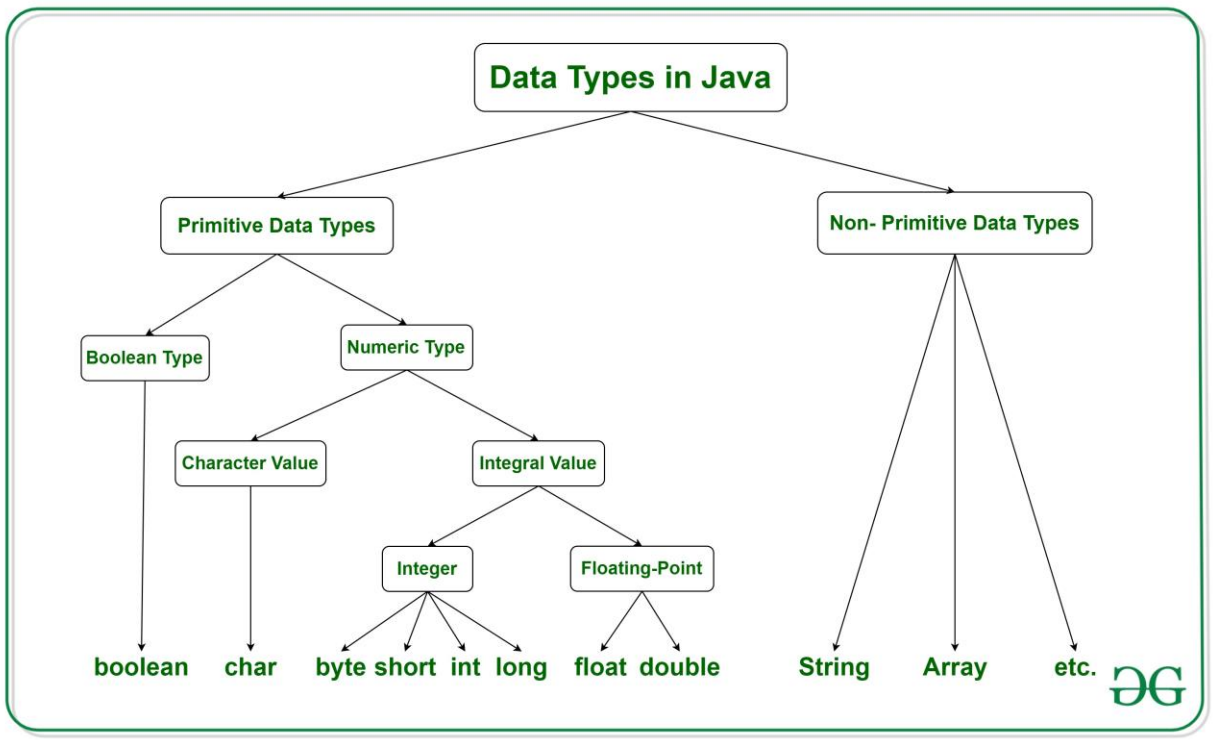
- The name of the class defined by the program HelloWorld should be same as name of file(HelloWorld.java).
- In Java, all codes must reside inside a class and there is at most one public class which contain main() method.
- By convention, the name of the main class(class which contain main method) should match the name of the file that holds the program.

# Java Comments

- In a program, comments take part in making the program become more human readable.
- By placing the detail of code involved and proper use of comments makes maintenance easier and finding bugs easily.
- Comments are ignored by the compiler while compiling a code.
- In Java there are **3 types of comments**:
  i. ***Single – line*** comments.
  ii. ***Multi – line*** comments.
  iii. ***Documentation*** comments.

# Java Data Types

- There are majorly two types of languages.
  - **Statically typed language:**
    - Each variable and expression type is already known at compile time.
    - Once a variable is declared to be of a certain data type, it cannot hold values of other data types.
    - ***Example:*** C, C++, Java.
  - **Dynamically typed languages:**
    - These languages can receive different data types over time.
    - ***Example:*** Ruby, Python
- Java is **statically typed and also a strongly typed language.**
  - Because in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language
  - And all constants or variables defined for a given program must be described with one of the data types.




## Primitive Data Types

| TYPE | DESCRIPTION | DEFAULT | SIZE | EXAMPLE LITERALS | RANGE OF VALUES |
|---|---|---|---|---|---|
| boolean | true or false | false | 1 bit | true, false | true, false |
| byte | twos complement integer | 0 | 8 bits | (none) | -128 to 127 |
| char | unicode character | \u0000 | 16 bits | 'a', '\u0041', '\101', '\\', '\','\n',' β' | character representation of ASCII values 0 to 255 |
| short | twos complement integer | 0 | 16 bits | (none) | -32,768 to 32,767 |
| int | twos complement integer | 0 | 32 bits | -2, -1, 0, 1, 2 | -2,147,483,648 to 2,147,483,647 |
| long | twos complement integer | 0 | 64 bits | -2L, -1L, 0L, 1L, 2L | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | IEEE 754 floating point | 0.0 | 32 bits | 1.23e100f, -1.23e-100f, .3f, 3.14F | upto 7 decimal digits |
| double | IEEE 754 floating point | 0.0 | 64 bits | 1.23456e300d, -1.23456e-300d, 1e1d | upto 16 decimal digits |

Notes:

- In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has value in the range [0, $2^{32}$-1]. Use the Integer class to use int data type as an unsigned integer.
- As above we can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}$-1. The Long class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.
- Both float and double data types were designed especially for scientific calculations, where approximation errors are acceptable. If accuracy is the most prior concern then, recommended not to use them and use ***BigDecimal*** class instead. For details: ***Rounding off errors in Java***.

## Non-Primitive / Reference Data Types

- The **Reference Data Types** will contain a memory address of variable value because the reference types won't store the variable value directly in memory.
- They are ***strings***, ***objects***, ***arrays***, etc.

### Strings:

- ***Strings*** are defined as an array of characters.
- The difference between a character array and a string is the string is terminated with a special character **'\0'**.

**Basic Syntax:**

```
<String_Type> <string_variable> = "<sequence_of_string>";
```

***Example:***

```
// Declare String without using new operator
String s = "GeeksforGeeks";

// Declare String using new operator
String s1 = new String("GeeksforGeeks");
```

### Class:

- A **Class** is a user-defined blueprint or prototype from which objects are created.
- It represents the set of properties or methods that are common to all objects of one type.
- In general, class declarations can include these components, in order:
  i. **Modifiers:** A class can be public or has default access (Refer this for details).
  ii. **Class name:** The name should begin with a initial letter (capitalized by convention).
  iii. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword ***extends***. A class can only extend (subclass) one parent.
  iv. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword ***implements***. A class can implement more than one interface.
  v. **Body:** The class body surrounded by braces, { }.

## Object:

- **Object**is a basic unit of Object-Oriented Programming and represents the real-life entities.
- A typical Java program creates many objects, which as we know, interact by invoking methods.
- An object consists of :
    i. **State:** It is represented by ***attributes*** of an object. It also reflects the ***properties*** of an object.
    ii. **Behavior:** It is represented by ***methods*** of an object. It also reflects the ***response*** of an object with other objects.
    iii. **Identity** : It gives a ***unique name*** to an object and enables one object to ***interact*** with other objects.

## Interface:

- Like a class, an **Interface** can have methods and variables.
- But the methods declared in an interface are by default ***abstract*** (only method signature, no body).
- Interfaces specify ***what a class must do and not how***. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.
- A Java library example is, ***Comparator Interface***, if a class implements this interface, then it can be used to sort a collection.

## Array:

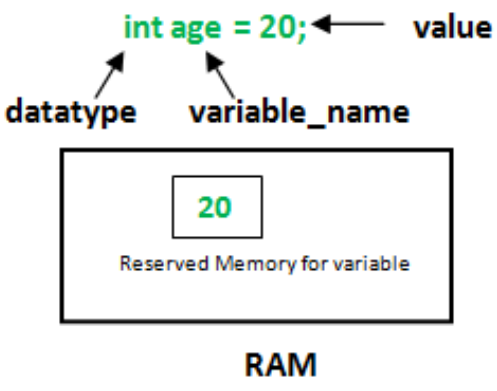
- ***Array*** is a ***group of like-typed variables*** that are referred to by a common name.
- Arrays in Java work differently than they do in C/C++.
- Following are some important point about Java arrays.
    - In Java all arrays are dynamically allocated. (discussed below).
    - Since arrays are objects in Java, we can find their length using member length.
    - This is different from C/C++ where we find length using sizeof.
    - A Java array variable can also be declared like other variables with [] after the data type.
    - The variables in the array are ordered and have an index beginning from 0.
    - Java array can be also be used as a static field, a local variable or a method parameter.
    - The **size** of an array must be specified by an int value and not long or short.
    - The direct superclass of an array type is ***Object***.
    - Every array type implements the interfaces ***Cloneable*** and ***java.io.Serializable***.

# Java Variables

### What is a Variable ?

- A variable is a name given to a memory location and is the basic unit of storage in a program.
- The value stored in a variable can be changed during program execution.
- It is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before use.

### How to declare variables ?




- **datatype:** Type of data that can be stored in this variable.
- **variable_name:** Name given to the variable.
- **value:** It is the initial value stored in the variable.

***Examples:***

```
float simpleInterest; //Declaring float variable
int time = 10, speed = 20; //Declaring and Initializing integer variable
char var = 'h'; // Declaring and Initializing character variable
```

# Types of Variables

## 1. Local Variables:

- A variable defined within a block or method or constructor is called local variable.
- These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.
- ***Initilisation*** of Local Variable is ***Mandatory***.

***Example:***

```
public class StudentDetails {
    public void StudentAge() {
        // local variable age
        int age = 0;
        age = age + 5;
        System.out.println("Student age is : " + age);
    }

    public static void main(String args[]) {
        StudentDetails obj = new StudentDetails();
        obj.StudentAge();
    }
}
```

**Output:**

```
Student age is : 5
```

Here, the variable age is a local variable to the function StudentAge(). If we use the variable age outside StudentAge() function, the compiler will produce an error

## 2. Instance Variables:

- Instance variables are ***non-static variables*** and are ***declared in a class outside any method, constructor or block***.
- These variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
- ***Initilisation*** of Instance Variable is ***NOT Mandatory***. Its default value is 0.
- Instance Variable can be accessed only by creating objects.
- In case we have multiple objects as in the below program, ***each object will have its own copies of instance variables***.

***Examples:***

```
import java.io.*;
class Marks {
    // These variables are instance variables, they are in a class and are not inside any function
    int engMarks;
    int mathsMarks;
}

class MarksDemo {
    public static void main(String args[]) {
        // first object
        Marks obj1 = new Marks();
        obj1.engMarks = 50;
        obj1.mathsMarks = 80;

        // second object
        Marks obj2 = new Marks();
        obj2.engMarks = 80;
        obj2.mathsMarks = 60;

        // displaying marks for first object
        System.out.println("Marks for first object:");
        System.out.println(obj1.engMarks);
        System.out.println(obj1.mathsMarks);

        // displaying marks for second object
        System.out.println("Marks for second object:");
        System.out.println(obj2.engMarks);
        System.out.println(obj2.mathsMarks);
    }
}
```

**Output:**

```
Marks for first object:
50
80
Marks for second object:
80
60
```

## 3. Static Variables

- Static variables are also Class variables and are declared similarly as instance variables.
- The difference is that static variables are declared using the ***static keyword*** within a class outside any method constructor or block.

- Unlike instance variables, we can ***only have one copy of a static variable*** per class irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- ***Initilisation*** of Static Variable is ***NOT Mandatory***. Its default value is 0.
- If we access the static variable like Instance variable (through an object), the compiler will show the warning message and it won't halt the program. The compiler will replace the object name to class name automatically.
- If we access the static variable without the class name, Compiler will automatically append the class name.
- No need to create an object of that class too access static variables, can simply access as `class_name.variable_name;`

***Example:***

```
import java.io.*;
class Emp {
  // static variable salary
  public static double salary;
  public static String name = "Harsh";
}

public class EmpDemo {
  public static void main(String args[]) {
    // accessing static variable without object
    Emp.salary = 1000;
    System.out.println(Emp.name + "'s average salary:"+ Emp.salary);
  }
}
```

**Output:**

```
Harsh's average salary:1000.0
```

# Java Keywords

## What are Java Keywords ?

- ***Reserved words*** in a language that are used for some internal process or represent some predefined actions.
- These words are therefore not allowed to use as a variable names or objects, using these will result into a ***compile time error***.

## List of reserved words or keywords

1. **abstract** -Specifies that a class or method will be implemented later, in a subclass
2. **assert** -Assert describes a predicate (a true–false statement) placed in a Java program to indicate that the developer thinks that the predicate is always true at that place. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort.
3. **boolean** – A data type that can hold True and False values only
4. **break** – A control statement for breaking out of loops
5. **byte** – A data type that can hold 8-bit data values
6. **case** – Used in switch statements to mark blocks of text
7. **catch** – Catches exceptions generated by try statements
8. **char** – A data type that can hold unsigned 16-bit Unicode characters
9. **class** -Declares a new class
10. **continue** -Sends control back outside a loop
11. **default** -Specifies the default block of code in a switch statement
12. **do** -Starts a do-while loop
13. **double** – A data type that can hold 64-bit floating-point numbers
14. **else** – Indicates alternative branches in an if statement
15. **enum** – A Java keyword used to declare an enumerated type. Enumerations extend the base class.
16. **extends** -Indicates that a class is derived from another class or interface
17. **final** -Indicates that a variable holds a constant value or that a method will not be overridden
18. **finally** -Indicates a block of code in a try-catch structure that will always be executed
19. **float** -A data type that holds a 32-bit floating-point number
20. **for** -Used to start a for loop
21. **if** -Tests a true/false expression and branches accordingly
22. **implements** -Specifies that a class implements an interface
23. **import** -References other classes
24. **instanceof** -Indicates whether an object is an instance of a specific class or implements an interface
25. **int** – A data type that can hold a 32-bit signed integer
26. **interface** – Declares an interface
27. **long** – A data type that holds a 64-bit integer
28. **native** -Specifies that a method is implemented with native (platform-specific) code
29. **new** – Creates new objects
30. **null** -Indicates that a reference does not refer to anything

31. **package** – Declares a Java package
32. **private** -An access specifier indicating that a method or variable may be accessed only in the class it's declared in
33. **protected** – An access specifier indicating that a method or variable may only be accessed in the class it's declared in (or a subclass of the class it's declared in or other classes in the same package)
34. **public** – An access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the class that defines it is accessible)
35. **return** -Sends control and possibly a return value back from a called method
36. **short** – A data type that can hold a 16-bit integer
37. **static** -Indicates that a variable or method is a class method (rather than being limited to one particular object)
38. **strictfp** – A Java keyword used to restrict the precision and rounding of floating point calculations to ensure portability.
39. **super** – Refers to a class's base class (used in a method or class constructor)
40. **switch** -A statement that executes code based on a test value
41. **synchronized** -Specifies critical sections or methods in multithreaded code
42. **this** -Refers to the current object in a method or constructor
43. **throw** – Creates an exception
44. **throws** -Indicates what exceptions may be thrown by a method
45. **transient** -Specifies that a variable is not part of an object's persistent state
46. **try** -Starts a block of code that will be tested for exceptions
47. **void** -Specifies that a method does not have a return value
48. **volatile** -Indicates that a variable may change asynchronously
49. **while** -Starts a while loop

### Reserved for future

Some keywords are reserved, even they are not currently in use.

- **const** -Reserved for future use
- **goto** – Reserved for future use

### Literals

They look like keywords, but in actual they are **literals** and still can't be used as identifiers in a program

- **true**
- **false**
- **null**

# Java Operators

### What are Java Operators ?

- Java provides many types of operators which can be used according to the need.
- They are classified based on the functionality they provide.

#### 1. Arithmetic Operators:

They are used to perform simple arithmetic operations on primitive data types.

- ***** Multiplication
- **/** Division
- **%** Modulo
- **+** Addition
- **–** Subtraction

#### 2. Unary Operators:

Unary operators need only one operand. They are used to increment, decrement or negate a value.

- **– Unary minus:** used for negating the values.
- **+ Unary plus:** used for giving positive values. Only used when deliberately converting a negative value to positive.
- **++ Increment operator:** used for incrementing the value by 1. There are two varieties of increment operator.
    - **Post-Increment :** Value is first used for computing the result and then incremented.
    - **Pre-Increment :** Value is incremented first and then result is computed.
- **— Decrement operator:** used for decrementing the value by 1. There are two varieties of decrement operator.
    - **Post-decrement :** Value is first used for computing the result and then decremented.
    - **Pre-Decrement :** Value is decremented first and then result is computed.
- **! Logical not operator:** used for inverting a boolean value.

### 3. Assignment Operator : '='

- Assignment operator is used to assign a value to any variable.
- It has a right to left associativity, i.e value given on right hand side of operator is assigned to the variable on the left and therefore right hand side value must be declared before using it or should be a constant.
- General format of assignment operator is,

```
variable = value;
```

- In many cases assignment operator can be combined with other operators to build a shorter version of statement called **Compound Statement**. For example, instead of a **=** a+5, we can write a **+=** 5.
    - **+=** for adding left operand with right operand and then assigning it to variable on the left.
    - **-=** for subtracting left operand with right operand and then assigning it to variable on the left.
    - ***=** for multiplying left operand with right operand and then assigning it to variable on the left.
    - **/=** for dividing left operand with right operand and then assigning it to variable on the left.
    - **%=** for assigning modulo of left operand with right operand and then assigning it to variable on the left.

### 4. Relational Operators :

- These operators are used to check for relations like equality, greater than, less than.
- They return boolean result after the comparison and are extensively used in looping statements as well as conditional if else statements.
- General format is:

```
variable relation_operator value
```

- Some of the relational operators are:
    - **== Equal to :** returns true of left hand side is equal to right hand side.
    - **!= Not Equal to :** returns true of left hand side is not equal to right hand side.
    - **< less than :** returns true of left hand side is less than right hand side.
    - **<= less than or equal to :** returns true of left hand side is less than or equal to right hand side.
    - **> Greater than :** returns true of left hand side is greater than right hand side.
    - **>= Greater than or equal to :** returns true of left hand side is greater than or equal to right hand side.

### 5. Logical Operators :

- These operators are used to perform "logical AND" and "logical OR" operation.
- One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e. it has a short-circuiting effect.
- Used extensively to test for several conditions for making a decision.
- Conditional operators are:
    - **&& Logical AND :** returns true when both conditions are true.
    - **|| Logical OR :** returns true if at least one condition is true.

### 6. Ternary operator :

- Ternary operator is a shorthand version of if-else statement.
- It has three operands and hence the name ternary.
- General format is:

```
condition ? if true : if false
```

### 7. Bitwise Operators :

- These operators are used to perform manipulation of individual bits of a number.
- They can be used with any of the integer types.
- They are used when performing update and query operations of Binary indexed tree.
    - **& Bitwise AND operator :** returns bit by bit AND of input values.
    - **| Bitwise OR operator :** returns bit by bit OR of input values.
    - **^ Bitwise XOR operator :** returns bit by bit XOR of input values.
    - **~ Bitwise Complement Operator :** This is a unary operator which returns the one's compliment representation of the input value, i.e. with all bits inversed.

### 8. Shift Operators :

- These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively.
- They can be used when we have to multiply or divide a number by two.

    - **« Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.
    - **» Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.
    - **»> Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.
- General Format:

```
number shift_op number_of_places_to_shift;
```

### 9. Instance of operator

- Used for type checking. It can be used to test if an object is an instance of a class, a subclass or an interface.
- General format :

```
object instance of class/subclass/interface
```

***Example:***

```
// Java program to illustrate instance of operator
class Operators {
    public static void main(String[] args) {
        Person obj1 = new Person();
        Person obj2 = new Boy();

        // As obj is of type person, it is not an instance of Boy or interface
        System.out.println("obj1 instanceof Person: " + (obj1 instanceof Person));
        System.out.println("obj1 instanceof Boy: " + (obj1 instanceof Boy));
        System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));

        // Since obj2 is of type boy, whose parent class is person and it implements
        // the interface Myinterface, it is instance of all of these classes
        System.out.println("obj2 instanceof Person: " + (obj2 instanceof Person));
        System.out.println("obj2 instanceof Boy: " + (obj2 instanceof Boy));
        System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
    }
}

class Person {
}

class Boy extends Person implements MyInterface {
}

interface MyInterface {
}
```

**Output:**

```
obj1 instanceof Person: true
obj1 instanceof Boy: false
obj1 instanceof MyInterface: false
obj2 instanceof Person: true
obj2 instanceof Boy: true
obj2 instanceof MyInterface: true
```

# Java Control Statements - Decision Making

### What are Java Control Statements ?

- A programming language uses control statements to control the flow of execution of program based on certain conditions.
- These are used to cause the flow of execution to advance and branch based on changes to the state of a program.
- These statements allows to control the flow of your program's execution based upon conditions known only during run time.

### 1. if

- Used to decide whether a certain statement or block of statements will be executed or not.

```
if(condition) {
   // Statements to execute if
   // condition is true
}
```

### 2. if-else

- We can use the else statement with if statement to execute a block of code when the condition is false.

```
if (condition){
    // Executes this block if condition is true
} else {
    // Executes this block if condition is false
}
```

### 3. nested-if

- It is an if statement that is the target of another if or else.

```
if (condition1) {
   // Executes when condition1 is true
   if (condition2) {
      // Executes when condition2 is true
   }
}
```

### 4. if-else-if ladder

- The if statements are executed from the top down.
- As soon as one if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.
- If none of the conditions is true, then the final else statement will be executed.

```
if (condition1) {
    // statement;
} else if (condition2) {
   // statement;
} else if (condition3) {
   // statement;
}
.
.
.
else {
   // statement;
}
```

### 5. Switch-case

- The switch statement is a multiway branch statement.
- It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

```
switch (expression){
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    break;
  .
  .
  case valueN:
    statementN;
    break;
  default:
    statementDefault;
}
```

- Expression can be of type byte, short, int, char, String or an enumeration.
- Dulplicate case values are not allowed.
- The default statement is optional.
- The break statement is used inside the switch to terminate a statement sequence.
- The break statement is optional. If omitted, execution will continue on into the next case.

### 6. Jump Statements

- Java supports 3 jump statements:
    - **break**
    - **continue**
    - **return**.
- These three statements transfer control to other part of the program.

# Java Loops

### What are Java Loops ?

- Facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.
- Java provides three ways for executing the loops.
- While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

### 1. while loop:

- Allows code to be executed repeatedly based on a given Boolean condition.
- The while loop can be thought of as a repeating if statement.

```
while (boolean condition) {
   // loop statements...
}
```

### 2. For Loop:

- Provides a concise way of writing the loop structure.
- Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line.
- Thereby providing a shorter, easy to debug structure of looping.

```
for (initialization condition; testing condition; increment/decrement){
    // statement(s)
}
```

### Enhanced For Loop

- Java also includes another version of for loop introduced in Java 5.
- Enhanced for loop provides a simpler way to iterate through the elements of a collection or array.
- It is inflexible and should be used only when there is a need to iterate through elements in a sequential manner without caring index.
- **Important Points:**
- The object/variable is immutable when enhanced for loop is used.
- It ensures that the values in the array can not be modified, so it can be said as a read-only loop.
    - Here we can't update the values as opposed to other loops where values can be modified.

```
for (T element : Collection obj or array) {
    // statement(s)
}
```

***Example:***

```
String array[] = { "Ron", "Harry", "Hermoine" };

for (String x : array) {
    System.out.println(x);
}
```

### Java For-Each loop

```
arrList.forEach((e) -> print(e));
```

### 3. do while loop

- Similar to while loop with only difference that it checks for condition after executing the statements.
- An example of ***Exit Control Loop.***

```
do {
    // statements..
} while (condition);
```

---

# Object Oriented Programming (OOPs) in Java

### What is OOPs in Java ?

- As the name suggests, Object-Oriented Programming or OOPs refers to languages that uses objects in programming.
- Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming.
- Provides means of structuring programs so that properties and behaviors are bundled into individual objects.
- **Main aim of OOP:** is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

### Advantages of object oriented programming:

1. **Improved software-development productivity:**
    - Provides improved software-development productivity over traditional procedure-based programming techniques due to below factors.
        a. **Modularity:** Provides separation of duties in object-based program development.
        b. **Extensibility:** Objects can be extended to include new attributes and behaviors.
        c. **Reusability:** Objects can also be reused within an across applications.
2. **Improved software maintainability:**
    - For the reasons mentioned above, it is easier to maintain.
    - Since the design is modular, part of the system can be updated in case of issues without a need to make large-scale changes.
3. **Faster development:**
    - Reuse enables faster development.
    - Come with rich libraries of objects, and code developed during projects is also reusable in future projects.

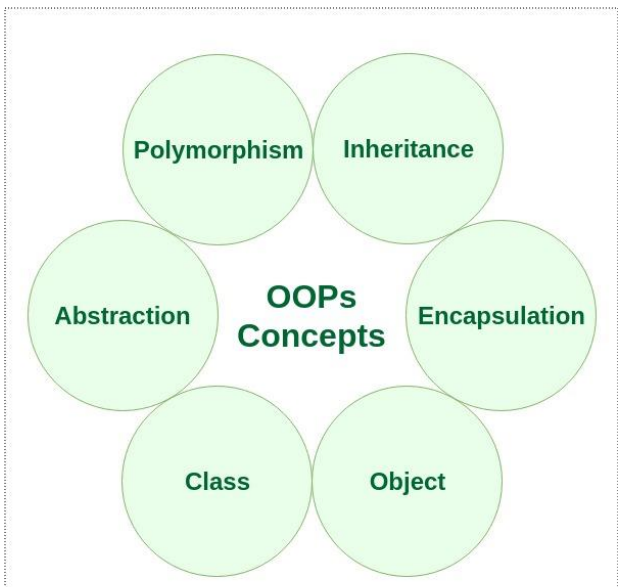
4. **Lower cost of development:**
    - Typically, more effort is put into the object-oriented analysis and design, which lowers the overall cost of development.
5. **Higher-quality software:**
    - Faster development of software and lower cost of development allows more time and resources to be used in the verification of the software.
    - Although quality is dependent upon the experience of the teams, it tends to result in higher-quality software.

### Disadvantages of object oriented programming:

- **Steep learning curve:**
    - The thought process involved may not be natural for some people, and it can take time to get used to it.
    - It is complex to create programs based on interaction of objects.
    - Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially
- **Larger program size:**
    - Typically involve more lines of code than procedural programs.
- **Slower programs:**
    - Typically slower than procedural programs, as they typically require more instructions to be executed.
- **Not suitable for all types of problems:**
    - There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style.
    - Applying object-oriented programming in those situations will not result in efficient programs.

## Major principles of object-oriented programming:

- **Classes, Objects (Instances), Methods**
- **Inheritance**
- **Polymorphism**
- **Encapsulation (Data Hiding)**
- **Abstraction (Detail Hiding)**




# Classes, Objects (Instances), Methods

## Class

- A class is a user defined blueprint or prototype from which objects are created.
- It represents the set of properties or methods that are common to all objects of one type.

### Components of Class

1. **Modifiers:**
    - A class can be public or has default access.
    - It ***can't be declared as private or protected.***
2. **Class name:**
    - The name should begin with a initial letter (capitalized by convention).
3. **Superclass (if any):**
    - The name of the class's parent (superclass), if any, preceded by the keyword **extends**.
    - A class can only extend (subclass) one parent.

4. **Interfaces (if any):**
    - A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword **implements**.
    - A class can implement more than one interface.
5. **Body:**
    - The class body surrounded by braces { }.

### Constructors:

- They are used for initializing new objects.
- Fields are variables that provides the state of the class and its objects.
- And methods are used to implement the behavior of the class and its objects.

**Note:** There are various types of classes that are used in real time applications such as `nested classes`, `anonymous classes`, `lambda expressions`.

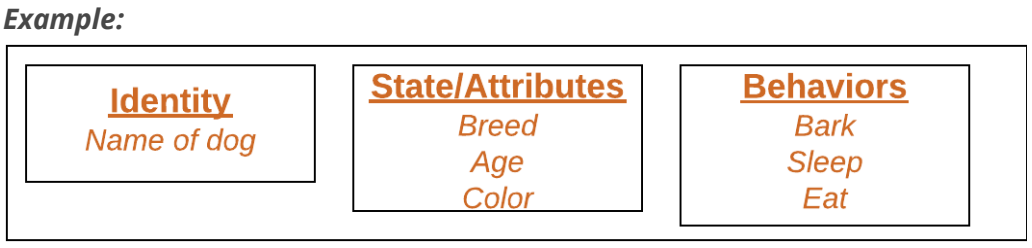
# Object

- It is a basic unit of Object Oriented Programming and represents the real life entities.
- A typical Java program creates many objects, which interacts by invoking methods.

## Components of Object:

1. **State:**
    - Represented by attributes of an object.
    - Also reflects the properties of an object.
2. **Behavior:**
    - Represented by methods of an object.
    - Also reflects the response of an object with other objects.
3. **Identity:**
    - It gives a unique name to an object and enables one object to interact with other objects.

***Example:***

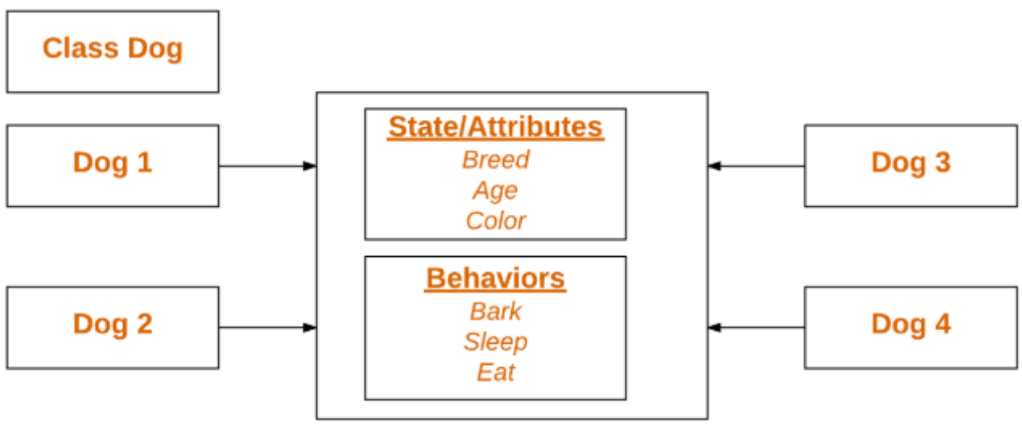



Notes:

- Objects correspond to things found in the real world.
- For example, a graphics program may have objects such as “circle”, “square”, “menu”.
- An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

## Declaring Objects / Instantiating Class

- When an object of a class is created, the class is said to be **instantiated**.
- All the instances share the attributes and the behavior of the class.
- But the values of those attributes, i.e. the state are unique for each object.
- A single class may have any number of instances.




Important Points:

- When we declare variables like `type name;` , it notifies the compiler that we will use this name to refer to data whose type is type.
- With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

- So for reference variable, type must be strictly a concrete class name.
- In general,we **can't** create objects of an abstract class or an interface.
- Simply declaring a reference variable does not create an object.
- If we declare reference variable(tuffy) like below, its value will be ***undetermined(null)*** until an object is actually created and assigned to it.

```
Dog tuffy;
```

## Initializing an object

- The **new operator** instantiates a class by allocating memory for a new object and returning a reference to that memory.
- The **new operator** also invokes the class constructor.

***Example:***

```
public class Dog {
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed, int age, String color){
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // method 1
    public String getName(){
        return name;
    }

    // method 2
    public String getBreed(){
        return breed;
    }

    // method 3
    public int getAge(){
        return age;
    }

    // method 4
    public String getColor(){
        return color;
    }

    @Override
    public String toString(){
        return("Hi my name is "+ this.getName() + ".\nMy breed, age and color are " +
               this.getBreed()+"," + this.getAge()+ ","+ this.getColor());
    }

    public static void main(String[] args){
        // Initializing a Dog object
        Dog tuffy = new Dog("tuffy","papillon", 5, "white");
        System.out.println(tuffy.toString());
    }
}
```

**Output:**

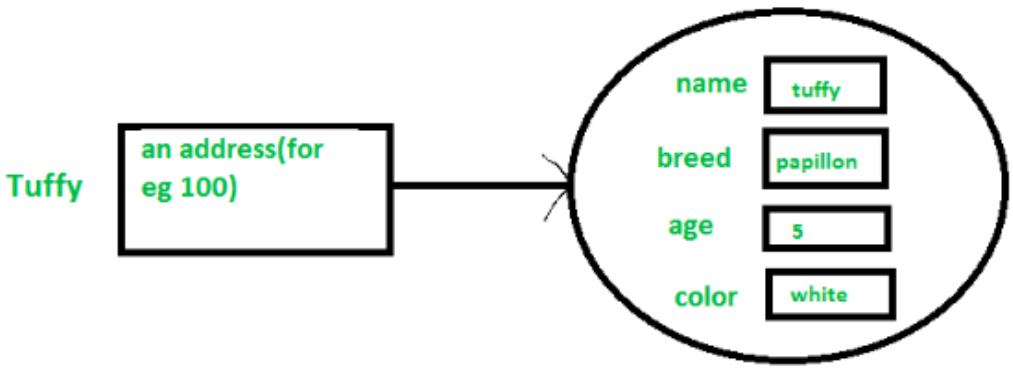
```
Hi my name is tuffy.
My breed,age and color are papillon,5,white
```

### About above class

- This class contains a single constructor.
- We can recognize a constructor because its declaration uses the same name as the class and it has no return type.
- The Java compiler differentiates the constructors based on the number and the type of the arguments.
- The constructor in the *Dog* class takes four arguments.
- The following statement provides "tuffy","papillon",5,"white" as values for those arguments:

```
Dog tuffy = new Dog("tuffy", "papillon", 5, "white");
```

- Result of above statement is:




Important Points:

- All classes have at least **one** constructor.
- If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, also called the default constructor.
- This default constructor calls the class parent's no-argument constructor (as it contain only one statement i.e super();), or the *Object* class constructor if the class has no other parent (as Object class is parent of all classes either directly or indirectly).

# Methods

- A method is a collection of statements that perform some specific task and return result to the caller.
- A method can perform some specific task without returning anything.
- Methods allow us to **reuse** the code without retyping the code.
- In Java, every method must be part of some class which is different from languages like C, C++ and Python.
- Java methods are strictly pass by value.

## Components of Method

1. **Access Modifier:**
   - Defines access type of the method i.e. from where it can be accessed in our application.
   - In Java, there 4 type of the access specifiers.
     - **public:** accessible in all class in the application.
     - **protected:** accessible within the package in which it is defined and in its **subclass(es)(including subclasses declared outside the package)**.
     - **private:** accessible only within the class in which it is defined.
     - **default (declared/defined without using any modifier):** accessible within same class and package within which its class is defined.
2. **The return type:**
   - The data type of the value returned by the method or void if does not return a value.
3. **Method Name:**
   - The rules for field names apply to method names as well, but the convention is a little different.
4. **Parameter list:**
   - Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis.
   - If there are no parameters, we must use empty parentheses ().
5. **Exception list:**
   - The exceptions we expect by the method can throw, we can specify these exception(s).
6. **Method body:**
   - The code we need to be executed to perform our intended operations enclosed between braces.

# Message Passing

- Objects communicate with one another by sending and receiving information to each other.
- A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results.
- Message passing involves specifying the name of the object, the name of the function and the information to be sent.

# Inheritance

### What is Inheritance ?

- It is mechanism by which one class is allowed to inherit the features(fields and methods) of another class.
- The keyword used for inheritance is **extends**.

Important terminologies:

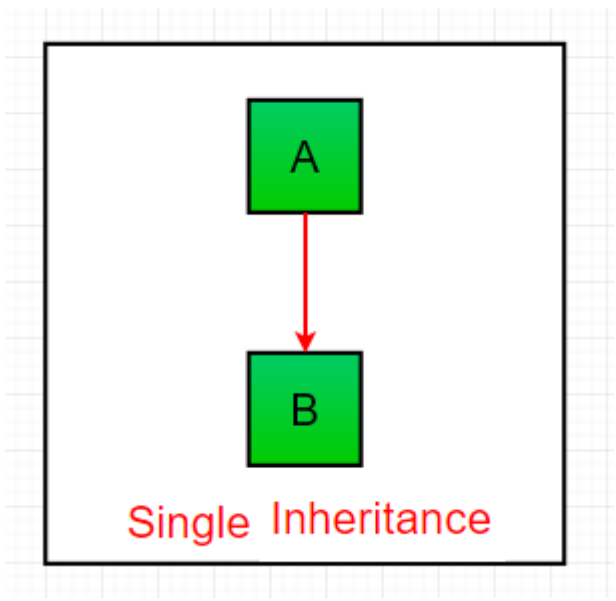
- **Super Class:**
  - The class whose features are inherited is known as superclass(or a base class or a parent class).
- **Sub Class:**
  - The class that inherits the other class is known as subclass(or a derived class, extended class, or child class).
  - The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:**
  - Inheritance supports the concept of “reusability”.
  - That means when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class.
  - By doing this, we are reusing the fields and methods of the existing class.

```
class DerivedClass extends BaseClass {
   //methods and fields
}
```

# Types of Inheritance

## 1. Single Inheritance :

- In single inheritance, subclasses inherit the features of one superclass.
- In image below, the class A serves as a base class for the derived class B.



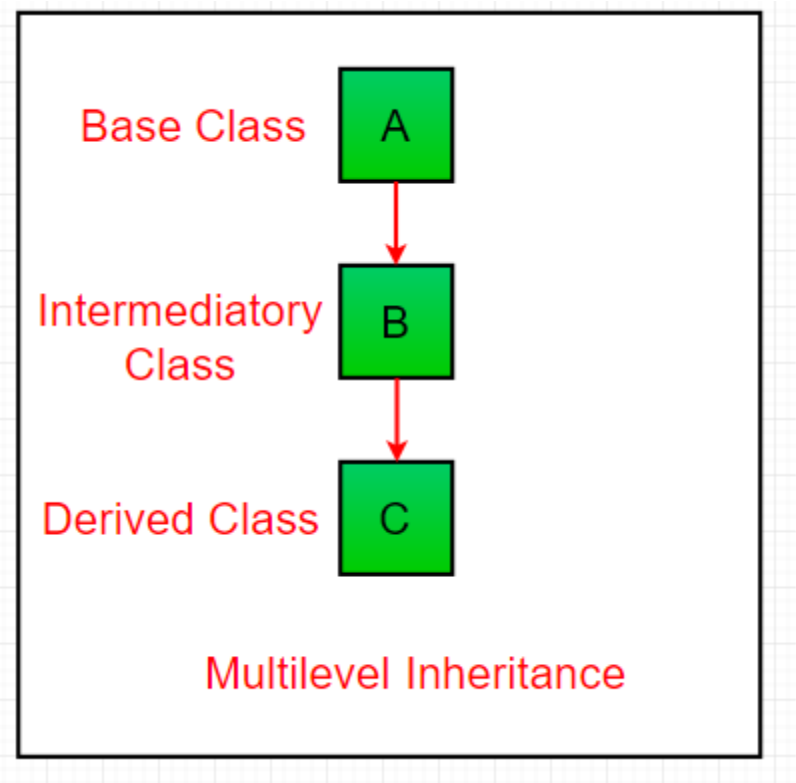

```
class one {
        public void print_geek() {
                System.out.println("Geeks");
        }
}

class two extends one {
        public void print_for() {
                System.out.println("for");
        }
}

public class Main {
        public static void main(String[] args) {
                two g = new two();
                g.print_for();
                g.print_geek();
        }
}
```

**Output:**

```
for
Geeks
```

## 2. Multilevel Inheritance

- In Multilevel Inheritance, a derived class will be inheriting a base class and the derived class also act as the base class to other class.
- In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.




```java
// Java program to illustrate the concept of Multilevel inheritance
class one {
        public void print_geek() {
                System.out.println("Geeks 1");
        }

  public void print_hello() {
                System.out.println("Hello from GrandParent");
        }
}

class two extends one {
        public void print_for() {
                System.out.println("for");
        }
}

class three extends two {
        public void print_geek() {
                System.out.println("Geeks 3");
        }
}

public class Main {
        public static void main(String[] args) {
                three g = new three();
    g.print_geek();
                g.print_for();
                g.print_hello();
        }
}
```

**Output:**

```
Geeks 3
for
Hello from GrandParent
```

## Method Resolution:

- Java starts finding the method from the current class and goes onto super classes one by one going level above.
- As all class is subclass of java **Object Class** it seraches till there and if not found gives ***cannot find symbol error***.

## Accessing Grandparent's member in Java using super

- In Java, a class cannot directly access the grandparent's members.

```java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent");
    }
}

class Parent extends Grandparent {
    public void Print() {
        System.out.println("Parent");
    }
}

class Child extends Parent {
    public void Print() {
        super.super.Print();  // Trying to access Grandparent's Print()
        System.out.println("Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

**Output:**

```
Compiler Error
```

### Important Points:

- There is error in line `super.super.print();`.
- In Java, a class cannot directly access the grandparent's members. It is allowed in C++ though.
- In C++, we can use scope resolution operator (::) to access any ancestor's member in inheritance hierarchy.
- ***In Java, we can access grandparent's members only through the parent class.***

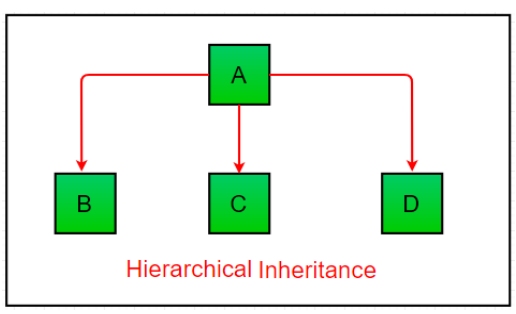
```
class Grandparent {
    public void Print() {
        System.out.println("Grandparent");
    }
}

class Parent extends Grandparent {
    public void Print() {
        super.Print();
        System.out.println("Parent");
    }
}

class Child extends Parent {
    public void Print() {
        super.Print();
        System.out.println("Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

**Output:**

```
Grandparent
Parent
Child
```

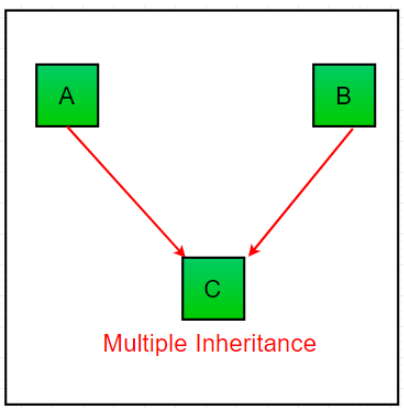
## 3. Hierarchical Inheritance :

- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.
- In below image, the class A serves as a base class for the derived class B, C and D.




## 4. Multiple Inheritance (Through Interfaces) :

- In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes.
- ***Please note that Java does not support multiple inheritance with classes.***
- ***In java, we can achieve multiple inheritance only through Interfaces.***
- In image below, Class C is derived from **interfaces A and B**.




```
interface one {
        public void print_geek();
}

interface two {
        public void print_for();
}

interface three extends one, two {
        public void print_geek();
```

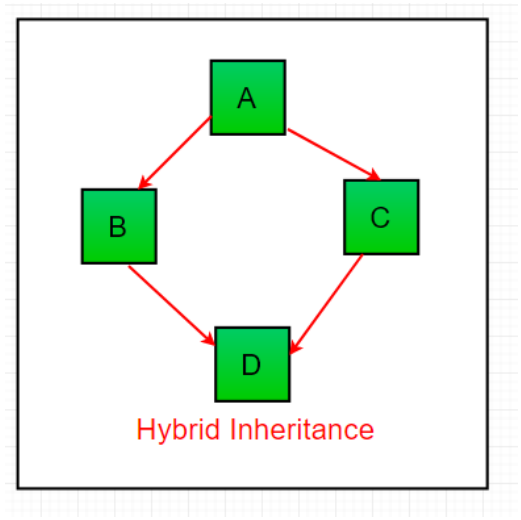
```
}

class child implements three {
        @Override
        public void print_geek() {
                System.out.println("Geeks");
        }

        public void print_for() {
                System.out.println("for");
        }
}

public class Main {
        public static void main(String[] args) {
                child c = new child();
                c.print_geek();
                c.print_for();
        }
}
```

**Output:**

```
Geeks
for
```

## 5. Hybrid Inheritance(Through Interfaces)

- It is a mix of two or more of the above types of inheritance.
- Since java doesn't support multiple inheritance with classes, the ***hybrid inheritance is also not possible with classes***.
- In java, we can ***achieve hybrid inheritance only through interfaces***.




### Important Points about inheritance in Java

- **Default superclass** :
  - Except ***Object*** class, which has no superclass, every class has one and only one direct superclass (single inheritance).
  - In the absence of any other explicit superclass, every class is implicitly a subclass of ***Object*** class.
- **Superclass can only be one :**
  - A superclass can have any number of subclasses.
  - But a subclass can have only ***one*** superclass.
  - This is because Java does not support ***multiple inheritance*** with classes.
  - Although with interfaces, multiple inheritance is supported by java.
- **Inheriting Constructors:**
  - ***A subclass inherits all the members (fields, methods, and nested classes) from its superclass.***
  - ***Constructors are not members, so they are not inherited by subclasses.***
  - But the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:**
  - A subclass does not inherit the private members of its parent class.
  - However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

### What all can be done in a Subclass ?

- In sub-classes we **can inherit members as is, replace them, hide them, or supplement them with new members**.
- The ***inherited fields can be used directly***, just like any other fields.
- We ***can declare new fields*** in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We ***can write a new instance method in the subclass that has the same signature*** as the one in the superclass by ***overriding*** it.
- We ***can write a new static method in the subclass that has the same signature*** as the one in the superclass, thus ***hiding*** it.
- We **can declare new methods** in the subclass that are not in the superclass.
- We ***can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super***.

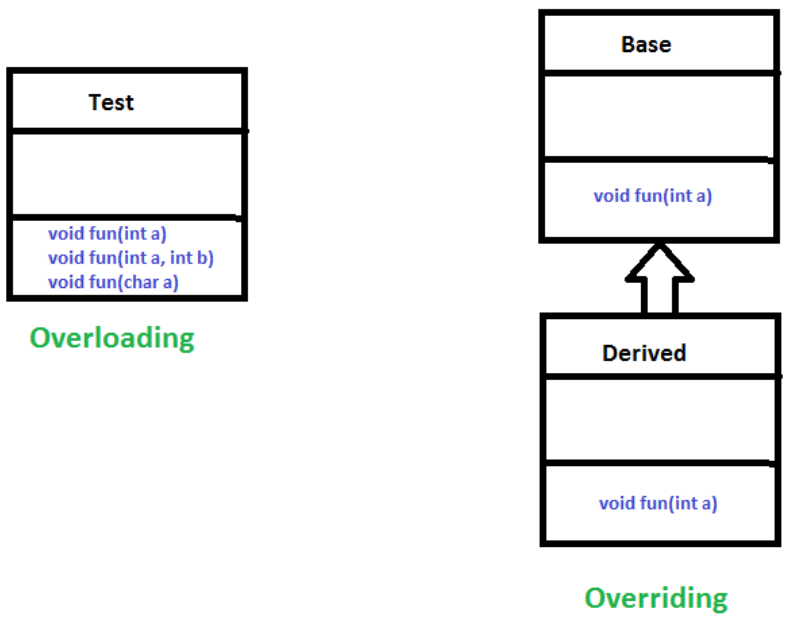
# Polymorphism

### What is Polymorphism ?

- Polymorphism allows us to perform a single action in different ways.
- Ability of OOPs programming languages to differentiate between entities with the same name efficiently.
- This is done by Java with the help of the signature and declaration of these entities.

### Types Polymorphism

- **Compile time Polymorphism**
  - **Method Overloading**
  - **Operator Overloading**
- **Runtime Polymorphism**
  - **Method Overriding**




## Compile Time Polymorphism

- It is also known as static polymorphism.
- This type of polymorphism is achieved by function overloading or operator overloading.

### Method Overloading

- When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**.
- Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

```java
// Overloaded sum with different number of parameters and differnt argument types
public class Sum {
  public int sum(int x, int y) {
    return (x + y);
  }

  public int sum(int x, int y, int z) {
    return (x + y + z);
  }

  public double sum(double x, double y) {
    return (x + y);
  }

  public static void main(String args[]) {
    Sum s = new Sum();
    System.out.println(s.sum(10, 20));
    System.out.println(s.sum(10, 20, 30));
    System.out.println(s.sum(10.5, 20.5));
  }
}
```

**Output:**

```
30
60
31.0
```

### Operator Overloading

- **In java, only "+" operator can be overloaded:**
  - To add integers
  - To concatenate strings

```java
class OperatorOVERDDN {
    void operator(String str1, String str2) {
        String s = str1 + str2;
        System.out.println("Concatinated String - " + s);
    }

    void operator(int a, int b) {
        int c = a + b;
        System.out.println("Sum = " + c);
    }
}

class Main {
    public static void main(String[] args) {
        OperatorOVERDDN obj = new OperatorOVERDDN();
        obj.operator(2, 3);
        obj.operator("joe", "now");
    }
}
```

**Output:**

```
Sum = 5
Concatinated String - joenow
```

# Runtime Polymorphism

- It is also known as Dynamic Method Dispatch.
- It is a process in which a function call to the overridden method is resolved at Runtime.
- This type of polymorphism is achieved by Method Overriding.

## Method Overriding

- Allows a subclass to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- That parent class function is said to be **overridden**.
- A superclass reference variable can refer to a subclass object k/a as **upcasting**.
- Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs k/a **dynamic dispatching**.

```java
class Parent {
    void show() {
        System.out.println("Parent's show()");
    }
}

class Child extends Parent {
    // This method overrides show() of Parent
    @Override
    void show() {
        System.out.println("Child's show()");
    }
}

class Main {
    public static void main(String[] args) {
        // If a Parent type reference refers to a Parent object, then Parent's show is called
        Parent obj;
        obj = new Parent();
        obj.show();

        // If a Parent type reference refers to a Child object, then Child's show() is called.
        // This is RUN TIME POLYMORPHISM.
        // Also known as Dynamic Dispatch (here child's show() method, above parent's show() method)
        obj = new Child();
        obj.show();
    }
}
```

**Output:**

```
Parent's show()
Child's show()
```

**Note:** In Java, we can override methods only, not the variables(data members), **so runtime polymorphism cannot be achieved by data members**.

```java
class A {
    int x = 10;
}

class B extends A {
    int x = 20;
}

public class Test {
    public static void main(String args[]) {
        A a = new B(); // object of type B

        // Data member of class A will be accessed
        System.out.println(a.x);
    }
}
```

**Output:**

```
10
```

## Why Method Overriding ?

- As stated earlier, overridden methods allow Java to support **run-time polymorphism**.
- Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the **"one interface, multiple methods"** aspect of polymorphism.
- **Dynamic Method Dispatch** is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness.
- The ability to exist code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.
- ***Overridden methods allow us to call methods of any of the derived classes without even knowing the type of derived class object.***

## Rules for method overriding

### 1. Overriding and Access-Modifiers :

- The **access modifier** for an overriding method can allow more, but not less, access than the overridden method.
- ***Example:*** a protected instance method in the super-class can be made public, but not private, in the subclass.
- Doing so, will generate compile-time error.

### 2. Private methods can not be overridden :

- **Private Methods** cannot be overridden as they are bonded during compile time.
- Therefore we can't even override private methods in a subclass.

```
class Parent {
    // private methods are not overridden
    private void m1() {
        System.out.println("From parent m1()");
    }

    protected void m2() {
        System.out.println("From parent m2()");
    }
}

class Child extends Parent {
    // new m1() method unique to Child class
    private void m1() {
        System.out.println("From child m1()");
    }

    // overriding method with more accessibility :- protected to public
    @Override
    public void m2() {
        System.out.println("From child m2()");
    }
}

class Main {
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        obj1.m2();
        Parent obj2 = new Child();
        obj2.m2();
    }
}
```

**Output:**

```
From parent m2()
From child m2()
```

### 3. Final methods can not be overridden :

- If we don't want a method to be overridden, we declare it as **final**.

```
class Parent {
    // Can't be overridden
    final void show() {}
}

class Child extends Parent {
    // This would produce error
    void show() {}
}
```

### 4. Static methods can not be overridden(Method Overriding vs Method Hiding) :

- When we defines a static method with same signature as a static method in base class, it is known as **Method Hiding**.
- Below table summarizes what happens when we define a method with the same signature as a method in a super-class.

| | SUPERCLASS INSTANCE METHOD | SUPERCLASS STATIC METHOD |
|---|---|---|
| SUBCLASS INSTANCE METHOD | Overrides | Generates a compile-time error |
| SUBCLASS STATIC METHOD | Generates a compile-time error | Hides |

### 5. The overriding method must have same return type (or subtype) :

- It is possible to have different return type but it should be sub-type of parent's return type.
- This phenomena is known as **covariant return type**.

### 6. Invoking overridden method from sub-class :

- We can call parent class method in overriding method using **super keyword**.

```
class Parent {
        void show() {
                System.out.println("Parent's show()");
        }
}

class Child extends Parent {
        // This method overrides show() of Parent
        @Override
        void show() {
                super.show();
                System.out.println("Child's show()");
        }
}

class Main {
        public static void main(String[] args) {
                Parent obj = new Child();
                obj.show();
        }
}
```

### 7. Overriding and constructor :

- We can not override constructor as parent and child class can never have constructor with same name.
- Constructor name must always be same as Class name.

### 8. Overriding and Exception-Handling :

- Below are two rules to note when overriding methods related to exception-handling.
    - **Rule#1:** If the super-class overridden method does not throws an exception, subclass overriding method can only throws the **unchecked exception**, throwing checked exception will lead to compile-time error.

```
class Parent {
    void m1() {
        System.out.println("From parent m1()");
    }

    void m2() {
        System.out.println("From parent  m2()");
    }
}

class Child extends Parent {
    @Override
    // no issue while throwing unchecked exception
    void m1() throws ArithmeticException {
        System.out.println("From child m1()");
    }

    @Override
    // compile-time error issue while throwing checked exception
    void m2() throws Exception {
        System.out.println("From child m2");
    }
}
```

    - **Rule#2:** If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in **Exception hierarchy** will lead to compile time error.Also there is no issue if subclass overridden method is not throwing any exception.

```
class Parent {
```

```
        void m1() throws RuntimeException {
            System.out.println("From parent m1()");
        }
    }

    class Child1 extends Parent {
        @Override
        // no issue while throwing same exception
        void m1() throws RuntimeException {
            System.out.println("From child1 m1()");
        }
    }
    class Child2 extends Parent {
        @Override
        // no issue while throwing subclass exception
        void m1() throws ArithmeticException {
            System.out.println("From child2 m1()");
        }
    }
    class Child3 extends Parent {
        @Override
        // no issue while not throwing any exception
        void m1() {
            System.out.println("From child3 m1()");
        }
    }
    class Child4 extends Parent {
        @Override
        // compile-time error issue while throwing parent exception
        void m1() throws Exception {
            System.out.println("From child4 m1()");
        }
    }
```

### 9. Overriding and abstract method:

- Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes.
- Otherwise a compile-time error will be thrown.

### 10. Overriding and synchronized/strictfp method :

- The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding.
- That means it's possible that a synchronized/strictfp method can override a non synchronized/strictfp one and vice-versa.

# Encapsulation (Data Hiding)

### What is Data Encapsulation ?

- It is defined as the wrapping up of data under a single unit, it is the mechanism that binds together code and the data it manipulates.
- In other words, it is a protective shield that prevents the data from being accessed by the code outside this shield.
- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- In encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

### How Encapsulation is achieved ?

- Encapsulation can be achieved by Declaring all the variables in the class as private.
- And then writing public methods in the class to set and get the values of variables.

```
public class Encapsulate {
    // private variables declared these can only be accessed by public methods of class
    private String geekName;
    private int geekRoll;
    private int geekAge;

    // get method for age to access private variable geekAge
    public int getAge() {
      return geekAge;
    }

    // get method for name to access private variable geekName
    public String getName() {
      return geekName;
    }

    // get method for roll to access private variable geekRoll
    public int getRoll() {
       return geekRoll;
```

```
    }

    // set method for age to access private variable geekage
    public void setAge( int newAge) {
      geekAge = newAge;
    }

    // set method for name to access private variable geekName
    public void setName(String newName) {
      geekName = newName;
    }

    // set method for roll to access private variable geekRoll
    public void setRoll( int newRoll)  {
      geekRoll = newRoll;
    }
}

public class TestEncapsulation {
    public static void main (String[] args) {
        Encapsulate obj = new Encapsulate();

        // setting values of the variables
        obj.setName("Harsh");
        obj.setAge(19);
        obj.setRoll(51);

        // Displaying values of the variables
        System.out.println("Geek's name: " + obj.getName());
        System.out.println("Geek's age: " + obj.getAge());
        System.out.println("Geek's roll: " + obj.getRoll());

        // Direct access of geekRoll is not possible due to encapsulation
        // System.out.println("Geek's roll: " + obj.geekName);
    }
}
```

**Output:**

```
Geek's name: Harsh
Geek's age: 19
Geek's roll: 51
```

### Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program
- **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

## Abstraction (Detail Hiding)

### What is Data Abstraction ?

- It is the property by virtue of which only the essential details are displayed to the user.
- The non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.
- It may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.
- ***Example:*** Consider a real-life example of a man driving a car.
    - The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car.
    - But he does not know about how on pressing the accelerator the speed is actually increasing.
    - He does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car.

### How Abstraction is achieved ?

- In java, abstraction is achieved by **interfaces** and **abstract classes**.
- We can achieve 100% abstraction using interfaces.

### Advantages of Abstraction:

- It reduces the complexity of viewing the things.
- Avoids code duplication and increases reusability.
- Helps to increase security of an application or program as only important details are provided to the user.

### Encapsulation vs Data Abstraction

- Encapsulation is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
- Encapsulation groups together data and methods that act upon the data but data abstraction deals with exposing the interface to the user and hiding the details of implementation.

# Abstract Classes

### What are abstract classes ?

- A class that ***contains one or more abstract methods*** and must be declared with **abstract keyword**.
- It is used to ***achieve abstraction to a certain extent*** the full abstraction is provided by **interfaces**.
- An abstract class may or may not have all abstract methods, some of them can be concrete methods
- A method defined abstract must always be redefined/overrrided in subclass or either make subclass itself abstract.
- There can be no object of an abstract class *i.e.* an abstract class can not be directly instantiated with ***new operator***.
- An abstract class can have parametrized constructors and default constructor is always present in an abstract class.

### What are abstract methods ?

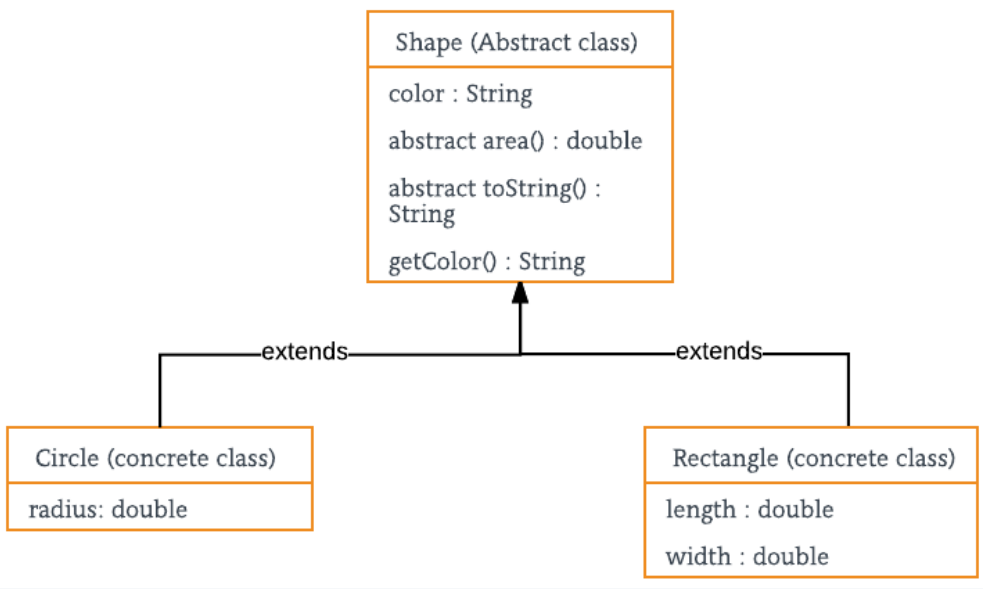
1. An abstract method is a method that is declared without an implementation.
2. It contains only signature of the mehod.

### When to use abstract classes and abstract methods ?

- There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

**Situation Example:**

- Consider a classic **"shape"** example, perhaps used in a computer-aided design system or game simulation.
- The base type is "shape" and each shape has a color, size and so on.
- From this, specific types of shapes are derived(inherited)-circle, square, triangle and so on – each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped.
- Some behaviors may be different, such as when we want to calculate the area of a shape.
- The type hierarchy embodies both the similarities and differences between the shapes.




```
abstract class Shape  {
    String color;

    // these are abstract methods
    abstract double area();
    public abstract String toString();

    // abstract class can have constructor
    public Shape(String color) {
        System.out.println("Shape constructor called");
        this.color = color;
    }

    // this is a concrete method
    public String getColor() {
        return color;
    }
}

class Circle extends Shape {
    double radius;

    public Circle(String color,double radius) {
```

```
        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override
    public String toString() {
        return "Circle color is " + super.color + "and area is : " + area();
    }
}

class Rectangle extends Shape{
    double length;
    double width;

    public Rectangle(String color,double length,double width) {
        // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

    @Override
    double area() {
        return length*width;
    }

    @Override
    public String toString() {
        return "Rectangle color is " + super.color + "and area is : " + area();
    }

}

public class Test  {
    public static void main(String[] args) {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

**Output:**

```
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Redand area is : 15.205308443374602
Rectangle color is Yellowand area is : 8.0
```

## Properties of Abstract Classes

**Property-1:** An instance of an abstract class cannot be created, we can have references of abstract class type though.

```
abstract class Base {
    abstract void fun();
}
class Derived extends Base {
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
        // Below line will cause compiler error as thetries to create an instance of abstract class.
        // Base b = new Base();

        // But we can have references of Base type.
        Base b = new Derived();
        b.fun();
    }
}
```

**Output:**

```
Derived fun() called
```

**Property-2:** Constructor of abstract class is called when an instance of a inherited class is created.

```
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
       Derived d = new Derived();
    }
}
```

**Output:**

```
Base Constructor Called
Derived Constructor Called
```

**Property-3:** We can have an abstract class without any abstract method, this allows to create classes that cannot be instantiated, but can only be inherited.

```
abstract class Base {
```

```
    void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base { }

class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.fun();
    }
}
```

**Output:**

```
Base fun() called
```

**Property-4:** Abstract classes can also have final methods (methods that cannot be overridden).

```
abstract class Base {
    final void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base {}

class Main {
    public static void main(String args[]) {
       Base b = new Derived();
       b.fun();
    }
}
```

**Output:**

```
Base fun() called
```

# Interfaces

## What is an Interface?

- Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
- Interfaces specify what a class must do and not how.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

**Basic Syntax:**

```
interface <interface_name> {
    // declare constant fields
    // declare methods that are abstract by default.
}
```

## Why do we use interface ?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve **multiple inheritance** .

### Important Points about Interfaces

- Interface can’t be instantiated but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface or interfaces (more than one interface) .
- A class that implements interface must implements all the methods in interface.
- ***All the methods are public and abstract, and all the fields are public, static, and final.***
- It is used to achieve **multiple inheritance** and **loose coupling**.

## Why use interfaces when we have abstract classes?

- The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

***Example:***

```
interface In1 {
    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}

// A class that implements the interface.
class TestClass implements In1 {
    // Implementing the capabilities of interface.
    public void display() {
        System.out.println("Geek");
    }

    public static void main (String[] args) {
        TestClass t = new TestClass();
        t.display();
```

```
        System.out.println(a);
    }
}
```

**Output:**

```
Geek
10
```

***A real World Example:***

```
interface Vehicle {
    // all are the abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

class Bicycle implements Vehicle{
    int speed;
    int gear;
     // to change gear
    @Override
    public void changeGear(int newGear){
        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){
        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){
        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed + " gear: " + gear);
    }
}

class Bike implements Vehicle {
    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){
        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){
        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){
        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed + " gear: " + gear);
    }

}
class GFG {
    public static void main (String[] args) {
        // creating an inatance of Bicycle doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");
        bicycle.printStates();

        // creating instance of the bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.println("Bike present state :");
        bike.printStates();
    }
}
```

**Output:**

```
Bicycle present state :
speed: 2 gear: 2
Bike present state :
speed: 1 gear: 1
```

## New features added in interfaces in Java 8

### 1. Default implementation for interface methods

Prior to Java 8, interface could not define implementation. We can now add default implementation for interface methods.

This default implementation has special use and does not affect the intention behind interfaces.

**Reason:**

Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

```
interface In1 {
    final int a = 10;
    default void display() {
        System.out.println("hello");
    }
}

class TestClass implements In1 {
    public static void main (String[] args) {
        TestClass t = new TestClass();
        t.display();
    }
}
```

**Output:**

```
hello
```

### 2. Defining static methods in interface

We can now define static methods in interfaces which can be called independently without an object.

**Note:-** these methods are not inherited.

```
interface In1 {
    final int a = 10;
    default void display() {
        System.out.println("hello");
    }
}

class TestClass implements In1 {
    public static void main (String[] args) {
        In1.display();
    }
}
```

**Output:**

```
hello
```

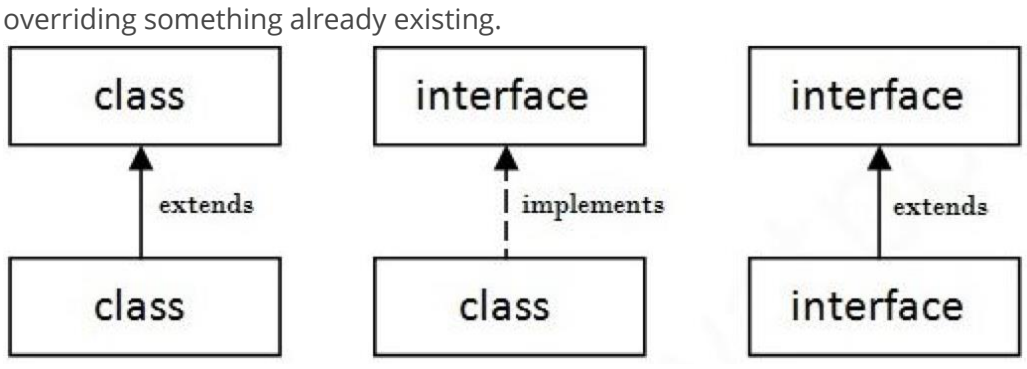
## New features added in interfaces in Java 9

From Java 9 onwards, interfaces can contain following also

1. Static methods
2. Private methods
3. Private Static methods

### Implements vs. Extends

- Both keywords are used when creating your own new class in the Java language.
- **Implements** is for when we are implementing an interface.
- **Extends** is for inheriting from a base class for extending something that already exists by adding more functionality to it or overriding something already existing.




- To solve problem of multiple-inheritance in Java we can only have one super class, but we can implement multiple interfaces.

---

# Arrays and Strings

This section discussed about the the 2 major concepts in Java programming language.

- Arrays
- Strings

## Arrays

### What are arrays in Java ?

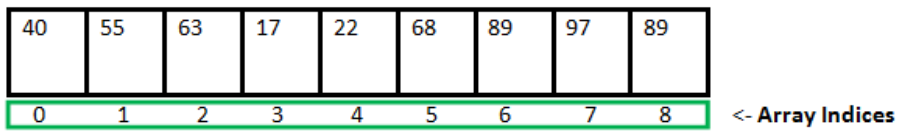
An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++.

Following are some important point about Java arrays.

- In Java all arrays are ***dynamically allocated***.
- Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- *The size of an array must be specified by an int value and not long or short.*
- The direct superclass of an array type is **Object**.
- Every array type implements the interfaces **Cloneable** and **java.io.Serializable**.

### Important Points

- Array can contains primitives data types as well as objects of a class depending on the definition of array.
- In case of primitives data types, the actual values are stored in ***contiguous memory locations***. In case of objects of a class, the actual objects are stored in ***heap segment***.




**Array Length = 9**
**First Index = 0**
**Last Index = 8**

## Creating / Declaring Array :

- The general form of a one-dimensional array declaration is:

```
type var-name[];
or
type[] var-name;
```

- We can create an array of any java data types either primitive data types, Boxed types, Object type, Collection Type or user defined data types:

```
// Array of Primitive data types
int[] intArr;
byte[] byteArr;
short[] shortArr;
long[] longArr;
float[] floatArr;
double[] doubleArr;
char[] charArr;

// Array of Boxed Types
Integer[] integerArr;

// Array of Object of unknown type
Object[] objArr;

// Array of Collection of unknown type
Collection[] collArr;

// Array of User Defined Type
MyClass[] myClassArr;

public static void main(String[] args){
  System.out.println("hello");
  System.out.println("hello");
  System.out.println("hello");
}
```

- Although the above first declaration establishes the fact that intArr is an array variable, **no array actually exists**.
- It simply tells to the compiler that this(intArr) variable will hold an array of the integer type.
- To link intArray with an actual, physical array of integers, we must allocate one using **new** and assign it to intArray.

## Instantiating / Initializing Array :

- When an array is declared, only a reference of array is created.
- To actually create or give memory to array, you create an array like this:

```
var-name = new type [size];
```

- To use *new* to allocate an array, **you must specify the type and number of elements to allocate.**

```
int intArr[];     //declaring array
intArr = new int[20];  // allocating memory to array
```

- We can even combine declaration and allocation of memory into a single statement.

```
int[] intArr = new int[20]; // combining both statements in one
```

- Elements in the array allocated by ***new*** will automatically be initialized to :
  - **zero** (for numeric types)
  - **false** (for boolean)
  - **null** (for reference types)
- Obtaining an array is a two-step process. First, we must declare a variable of the desired array type. Second, we must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, **in Java all arrays are dynamically allocated.**

### Array Literal:

- In a situation, where the size of the array and variables of array are already known, array literals can be used.

```
 int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 }; // Declaring array literal
```

- The length of this array determines the length of the created array.
- There is no need to write the new int[] part in the latest versions of Java.

```
 int[] intArray = { 1,2,3,4,5,6,7,8,9,10 }; // Declaring array literal in newer Java Versions
```

## Accessing Array :

- Each element in the array is accessed via its index.
- The index begins with 0 and ends at (total array size)-1.
- All the elements of array can be accessed using Java for Loop.

```
// accessing the elements of the specified array
for (int i = 0; i < arr.length; i++) {
  System.out.println("Element at index " + i + " : "+ arr[i]);
}
```

***Example:***

```
class Student {
    public int roll_no;
    public String name;

    Student(int roll_no, String name) {
        this.roll_no = roll_no;
        this.name = name;
    }
}

// Elements of array are objects of a class Student.
public class UserDefinedDataTypeArrayExample {
    public static void main (String[] args) {
        // declares an Array of of type Student.
        Student[] arr;

        // allocating memory for 5 objects of type Student.
        arr = new Student[5];

        // initialize the first elements of the array
        arr[0] = new Student(1,"aman");

        // initialize the second elements of the array
        arr[1] = new Student(2,"vaibhav");
        arr[2] = new Student(3,"shikar");
        arr[3] = new Student(4,"dharmesh");
        arr[4] = new Student(5,"mohit");

        // accessing the elements of the specified array
        for (int i = 0; i < arr.length; i++)
            System.out.println("Element at " + i + " : " + arr[i].roll_no + " " + arr[i].name);
    }
}
```

**Output:**

```
Element at 0 : 1 aman
Element at 1 : 2 vaibhav
Element at 2 : 3 shikar
Element at 3 : 4 dharmesh
Element at 4 : 5 mohit
```

## Multi-dimensional Arrays :

- Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array also k/a **Jagged Arrays**.
- A multidimensional array is created by appending ***one set of square brackets ([]) per dimension***.

```java
int[][] int2DArr = new int[10][20];  // 2D array or matrix
int[][][] int3DArr = new int[10][20][10];  // 3D array
```

**Example:**

```java
class multiDimensionalArrayExample {
    public static void main(String args[]) {
        // declaring and initializing 2D array
        int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

        // printing 2D array
        for (int i=0; i< 3 ; i++) {
            for (int j=0; j < 3 ; j++) {
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

**Output:**

```
2 7 9
3 6 1
7 4 2
```

# Other Array Concepts :

### 1. Passing Arrays to Methods

- Like variables, we can also pass arrays to methods.
- *Example:* below program pass array to method *sum* for calculating sum of array's values.

```java
class Test {
    public static void main(String args[])  {
        int arr[] = {3, 1, 2, 5, 4};

        sum(arr);  // passing array to method m1
    }

    public static void sum(int[] arr) {
        int sum = 0;  // getting sum of array values

        for (int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }

        System.out.println("sum of array values : " + sum);
    }
}
```

**Output:**

```
sum of array values : 15
```

### 2. Array Members

- Arrays are ***object of a class*** and ***direct superclass of arrays is class Object***.
- The members of an array type are all of the following:
    - `public final length` which contains the number of components of the array, length may be positive or zero.
    - All the members and methods ar inherited from class Object.
    - The only method of Object that is not inherited is its clone method.
    - `public clone()`, which overrides clone method in class Object and throws no checked exceptions.

# Arrays Class :

### What is Arrays Class in Java ?

- It is in **java.util package** and is a part of the Java Collection Framework.
- This class provides static methods to dynamically create and access Java arrays.
- It consists of only static methods and the methods of Object class.
- The methods of this class can be used by the class name itself.

### Class Hierarchy :

```
java.lang.Object
  ↳ java.util.Arrays
```

### Class Declaration :

```java
public class Arrays extends Object
```

### Basic Syntax:

```java
Arrays.<methodName>;
```

### Need for Java Arrays Class

- There are often times when **loops** are used to do some tasks on an array like:
  - Fill an array with a particular value.
  - Sort an Arrays.
  - Search in an Arrays.
  - And many more.
- *Arrays class provides several static methods that can be used to perform these tasks directly without the use of loops.*

## Methods in Arrays Class :

- `toString(arr):` returns a String representation of the contents of this Arrays.
- `asList(arr):`: returns a fixed-size list backed by the specified Arrays.
- `binarySearch(arr, key):` searches for the key in the array with the help of Binary Search algorithm.
- `binarySearch(arr, fromIndex, toIndex, key, comparator):` searches key in the specified range in the arr.
- `fill(arr, newVal):` assigns this newVal to each element of this Array.
- `fill(arr, fromIndex, toIndex, newVal):` assigns the new value in given range.
- `equals(arr1, arr2):` checks if both the arrays are equal or not.
- `sort(arr):` sorts the complete array in ascending order.
- `sort(arr, fromIndex, toIndex, comparator):` sorts the array in range, if comparator provided then a/c to it or else ascending order.
- `stream(arr):` returns a sequential stream with the specified array as its source.

## New Java 9 Methods:

- `mismatch(arr1, arr2):` finds and returns the index of the first unmatched element between the two arrays.
- `compare(arr1, arr2):` compares two arrays lexicographically.

### Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Ex3ArraysClassOperations {
    public static void main(String[] args){
        // Create an array using :=> Type[] arr = new Type[size]
        int[] arr = new int[5];

        // Filling values in array using : arr[index] = val
        arr[0] = 10; arr[1] = 20; arr[2] = 15;
        arr[3] = 22; arr[4] = 35;

        // Get the length of the arr using : length member
        System.out.println("Length of the arr : " + arr.length);

        // Get string representation of array using : toString(arr)
        String arrString = Arrays.toString(arr);
        System.out.println("String rep of arr : " + arrString);

        // List representation of array using : asList(arr)
        System.out.println("List representation of arr : " + Arrays.asList(arr));

        // Binary Search in arr using : binarySearch(arr, key)
        Arrays.sort(arr);
        System.out.println("Find key 22 in arr : " + Arrays.binarySearch(arr, 22));

        // Binary Search in range using : binarySearch(arr, fromIndex, toIndex, key)
        System.out.println("Find key 22 in arr : " + Arrays.binarySearch(arr, 1, 3, 22));

        // Fill the arr with new value using :=> fill(arr, newVal)
        Arrays.fill(arr, 55);
        System.out.println("Arrays after filling with 55 : " + arr);

        int arr1[] = { 10, 20, 15, 22, 35 };
        int arr2[] = { 10, 15, 22 };

        // Check if arrays are equal using :=> equals(arr1, arr2)
        System.out.println("Are arr1 and arr2 equal ? : " + Arrays.equals(arr1, arr2));

        // sort arr using :=> sort(arr)
        Arrays.sort(arr1);
        System.out.println("Sorted Arr : " + Arrays.toString(arr1));

        // Use stream using :=> stream(arr)
        List<Integer> arrIntegerList = Arrays.stream(arr1)   // Stream of int
                                             .boxed()    // Stream of Integer
                                             .collect(Collectors.toList());
        System.out.println("Integer List of arr : " + arrIntegerList);
    }
}
```

### Output:

```
Length of the arr : 5
String rep of arr : [10, 20, 15, 22, 35]
List representation of arr : [[I@74a14482]
Find key 22 in arr : 3
Find key 22 in arr : -4
Arrays after filling with 55 : [I@74a14482
Are arr1 and arr2 equal ? : false
Sorted Arr : [10, 15, 20, 22, 35]
Integer List of arr : [10, 15, 20, 22, 35]
```

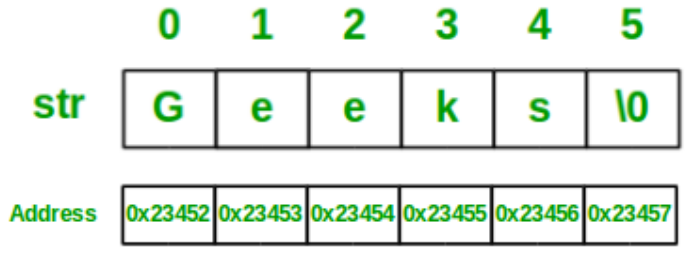
# Strings

### What are Strings in Java ?

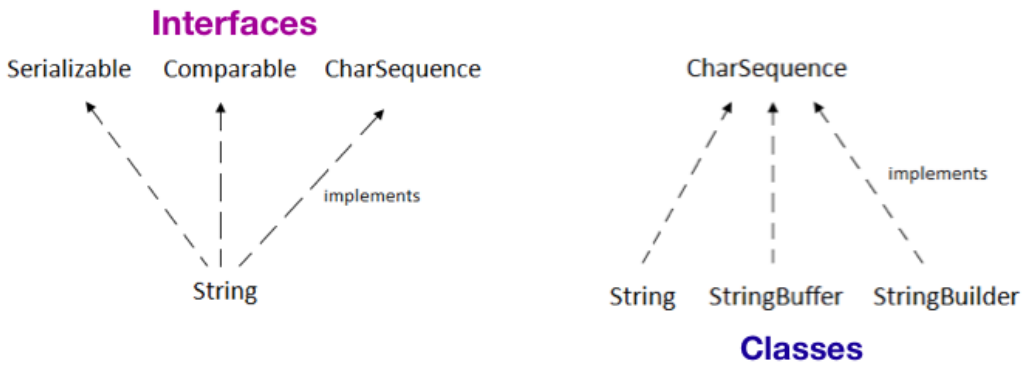
- Strings in Java are Objects that are backed internally by a char array.
- Since arrays are immutable(cannot grow), Strings are immutable as well.
- Whenever a change to a String is made, an entirely new String is created.

### Basic Syntax:

```
String stringVarName = "sequence of chars or words"

// Example:
String str = "Geeks";
```




Interfaces and Classes




### CharSequence Interface :

- It is used to represent the sequence of characters.
- **String**, **StringBuffer** and **StringBuilder** classes implement it. It means, we can create strings in java by using these three classes.

## 1. String

- String is a sequence of characters.
- In java, objects of String are immutable which means a constant and cannot be changed once created.

### Constructors :

```
String str = "ExampleOfString";
String str = new String ("ExampleOfString");
```

### Methods :

- `length()` - Returns the length or number of characters.
- `isEmpty()` - Checks if it is empty.
- `charAt(i)` - Returns the character at **i**th index.
- `substring(i)` - Returns the substirng form **i**th character to end.
- `substring(i, j)` - Returns the substirng form **i**th to (j-1)th character.
- `subsequence(i, j)` - returns the subsequence b/w i and j.
- `concat(s)` - Concatenates given string to end.
- `indexOf(s)` - Index of first occurrence.
- `indexOf(s, i)` - Index of first occurrence starting from given index.
- `lastIndexOf(s)` - Last index of the given string.
- `equals(obj)` - Compares string with given object.
- `equalsIgnoreCase(anotherStr)` - Compares string to another string igonoring case.
- `contentEquals(content)` - to check if content from buffer or builder equals to it.
- `compareTo(anotherStr)` - Compares 2 strings lexicographically.
- `compareToIgnoreCase(anotherStr)` - Compares 2 strings lexicographically ignoring case.
- `contains(anotherStr)` - checks if it contains the given str.
- `startsWith(str)` and `startsWith(str, offset)` to check if it starts with givn str.
- `endsWith(str)` - to check if it endswith given str.
- `matches(regex)` - to check if it matches with regex.

- `split(regex):` - splits using regex.
- `toLowerCase():` - Converts all characters in string to lower case.
- `toUpperCase():` - Converts all characters in string to upper case.
- `trim():` - Removes whitespaces at both ends but not in middles.
- `replace(oldChar, newChar):` - Replaces all occurences of old char with new in the string.
- `replaceAll(regex, replacement):` and `replaceFirst(regex, replacement):` - to replace within the string.

***Example:***

```java
import java.util.Arrays;

public class StringCreationAndOperations {
    public static void main(String[] args){
        // Create string using :=> literals
        String s = "Astik Anand";

        // Get count of characters using :=> length()
        System.out.println("Length of string s : " + s.length());

        // Check if it is empty using :=> isEmpty()
        System.out.println("Is string empty ? : " + s.isEmpty());

        // Get the char at ith position using :=> charAt(i)
        System.out.println("Char at 4th index : " + s.charAt(4));

        // Get the substring from ith to end using : s.substring(i)
        System.out.println("Substring from 4th index to end : " + s.substring(4));

        // Get the substring from ith to (j-1)th using : substring(i, j)
        System.out.println("Substring from 4th to 9th : " + s.substring(4, 9));

        // Get the subsequence b/w i and j-1 using :=> subsequence(i, j)
        System.out.println("Subsequence form 6th to 9th : " + s.subSequence(6, 9));

        // Concatenate given string to end using :=> s.concat(anotherStr)
        String s1 = "Hello! ";
        String s2 = "Folks, Welcome";
        System.out.println("Concatenated String : " + s1.concat(s2));

        // Get index of first occurrence using :=> indexOf(str)
        String s3 = "Learn Share Learn";
        System.out.println("Index of \"Share\" : " + s3.indexOf("Share"));

        // Get index of first occurrence starting at ith index using :=> indexOf(str, i)
        System.out.println("Index of 'a' starting at 3 : " + s3.indexOf('a', 3));

        // Get last index using :=> lastIndexOf(str)
        System.out.println("Last index of \"Learn\" : " + s3.lastIndexOf("Learn"));

        // Check equality of Strings using :=> equals(anotherStr)
        String s4 = "WeLcOMe";
        System.out.println("Is \"WeLcOMe\" equal to \"Welcome\" ? : " + s4.equals("Welcome"));

        // Check equality of Strings ignoring case using :=> equalsIgnoreCase(anotherStr)
        System.out.println("Is \"WeLcOMe\" equal to \"Welcome\" after ignoring case ? : "
                + s4.equalsIgnoreCase("Welcome"));

        // Check if content equals if content from StringBuffer or Builder using :=> contentEquals(content)
        System.out.println("Is \"WeLcOMe\" content equal to \"Welcome\" ? : "
                + s4.contentEquals(new StringBuffer("Welcome")));

        // Compare String Lexicographically using :=> compareTo(anotherStr)
        String s5 = "abcde";
        System.out.println("Comparing \"abcde\" to \"abcdb\" : " + s5.compareTo("abcdb"));

        // Check if string contains given str using :=> contains(str)
        System.out.println("Does s5 contains \"cd\" ? : " + s5.contains("cd"));

        // Check if string starts with given str using :=> startsWith(str)
        System.out.println("Does s5 startsWith \"abc\" ? : " + s5.startsWith("abc"));

        // Check if string ends with given str usign :=> endsWith(str)
        System.out.println("Does s5 startsWith \"cde\" ? : " + s5.endsWith("cde"));

        // Split on the basis of regex using :=> split(regex)
        String s6 = "I am   a   Programmer  ";
        String[] splitted = s6.split("\\s+");
        System.out.println("Split using regex : " + Arrays.toString(splitted));

        // Convert to Lower Case using :=> toLowerCase()
        String s7 = "ProGraMMeR";
        System.out.println("Lower case of \"ProGraMMeR\" : " + s7.toLowerCase());

        // Convert to Upper Case using :=> toUpperCase()
        System.out.println("Upper case of \"ProGraMMeR\" : " + s7.toUpperCase());

        // Trim spaces using :=> trim()
        String s8 = " Learn Programming   ";
        System.out.println("After trim : " + s8.trim());

        // Replace characters using : replace(oldChar, newChar)
        String s9 = "abracadabra";
        System.out.println("Replace 'a' with 'm' in \"abracadabra\" : " + s9.replace('a', 'm'));

        // Replace using regex : replaceFirst(regex, replacement)
        String s10 = "java, read, job, practice, senior, java, write, job, senior";
        String regex = "(java|job|senior)";
        System.out.println("Replace s10 using replaceFirst(regex, \"XYZ\") : "
                + s10.replaceFirst(regex , "XYZ"));

        // Replace using regex : replaceAll(regex, replacement)
        System.out.println("Replace s10 using replaceAll(regex, \"XYZ\") : "
                + s10.replaceAll(regex, "XYZ"));
```

```
    }
}
```

**Output:**

```
Length of string s : 11
Is string empty ? : false
Char at 4th index : k
Substring from 4th index to end : k Anand
Substring from 4th to 9th : k Ana
Subsequence form 6th to 9th : Ana
Concatenated String : Hello! Folks, Welcome
Index of "Share" : 6
Index of 'a' starting at 3 : 8
Last index of "Learn" : 12
Is "WeLcOMe" equal to "Welcome" ? : false
Is "WeLcOMe" equal to "Welcome" after ignoring case ? : true
Is "WeLcOMe" content equal to "Welcome" ? : false
Comparing "abcde" to "abcdb" : 3
Does s5 contains "cd" ? : true
Does s5 startsWith "abc" ? : true
Does s5 startsWith "cde" ? : true
Split using regex : [I, am, a, Programmer]
Lower case of "ProGraMMeR" : programmer
Upper case of "ProGraMMeR" : PROGRAMMER
After trim : Learn Programming
Replace 'a' with 'm' in "abracadabra" : mbrmcmdmbrm
Replace s10 using replaceFirst(regex, "XYZ") : XYZ, read, job, practice, senior, java, write, job, senior
Replace s10 using replaceAll(regex, "XYZ") : XYZ, read, XYZ, practice, XYZ, XYZ, write, XYZ, XYZ
```

# 2. StringBuffer

- StringBuffer is a peer class of String that provides much of the functionality of strings.
- ***String represents fixed-length***, immutable character sequences.
- ***StringBuffer represents growable and writable character sequences***.
- It may have characters and substrings inserted in the middle or appended to the end.
- It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed.

### Important Points :

- String buffers are `thread-safe` for use by multiple threads. The methods can be synchronized wherever necessary so that all the operations on any particular instance behave as if they occur in some serial order.
- Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence) this class synchronizes only on the string buffer performing the operation, not on the source.

### Constructors :

```
StringBuffer s1 = new StringBuffer(); // reserves room for 16 characters without reallocation
StringBuffer s2 = new StringBuffer(20); // acceepts number to set the size of buffer explicitly
StringBuffer s3 = new StringBuffer("ExampleOfStringBuffer"); // sets the initial contents and reserves
                                                             // room for 16 more characters without reallocation
```

### Methods :

- `length():` - gives lenght of buffer and `capacity():` gives the capacity of the buffer.
- `append(item):` - appends item at end of existing string, item can be string, int.
- `equals(obj)` - to check if it equals to given obj.
- `indexOf(str):` - gives index of the first occurrence.
- `indexOf(str, index):` - gives index of the first occurrence starting from index.
- `lastIndexOf(str):` - gives index of the last occurrence.
- `insert(index, item):` - inserts item at given index, item can be string, int, float, double, char etc.
- `charAt(index):` - returns char at the given index.
- `setCharAt(index, newChar):` - Sets the new char at given index.
- `reverse():` - reverses the characters in the buffer.
- `replace(start, end, anotherStr):` - replaces all characters from start to end with antotherStr.
- `delete(start, end):` and `deleteCharAt(index):` - to delete characters.
- `substring(i):` - and `substring(i, j):` - to get the substring from i to end and from i to j-1.
- `subsequence(i, j):` - get subsequence from i to j-1.
- `toString():` - returns a string representing the data in this sequence.

# 3. StringBuilder

- Represents a mutable sequence of characters.
- String Class in Java creates and immutable sequence of characters.
- StringBuilder class provides an alternate to String Class, as it creates a mutable sequence of characters.

### Why StringBuilder after StringBuffer ?

- Java5 added a new string class to Java's already powerful string handling capabilities k/a **StringBuilder**.

- It is identical to StringBuffer except it is not synchronized, which means that it is `NOT thread-safe`.
- The advantage of StringBuilder is **faster performance**.
- However, in cases in which you are using multithreading, you must use StringBuffer rather than StringBuilder.

### Constructors :

```
StringBuilder s1 = new StringBuilder(); // Creates with initial capacity of 16 characters.
StringBuilder s1 = new StringBuilder(20); // Accepts number to set the size explicitly
StringBuilder s1 = new StringBuilder(seq); // Contains the same characters as the specified seq.
StringBuilder s1 = new StringBuilder(str); // Initialized to the contents of the specified string.
```

### Methods :

Has all the methods same as of StringBuffer.

- `length():` - gives lenght of buffer and `capacity():` gives the capacity of the buffer.
- `append(item):` - appends item at end of existing string, item can be string, int.
- `equals(obj)` - to check if it equals to given obj.
- `indexOf(str):` - gives index of the first occurrence.
- `indexOf(str, index):` - gives index of the first occurrence starting from index.
- `lastIndexOf(str):` - gives index of the last occurrence.
- `insert(index, item):` - inserts item at given index, item can be string, int, float, double, char etc.
- `charAt(index):` - returns char at the given index.
- `setCharAt(index, newChar):` - Sets the new char at given index.
- `reverse():` - reverses the characters in the buffer.
- `replace(start, end, anotherStr):` - replaces all characters from start to end with antotherStr.
- `delete(start, end):` and `deleteCharAt(index):` - to delete characters.
- `substring(i):` - and `substring(i, j):` - to get the substring from i to end and from i to j-1.
- `subsequence(i, j):` - get subsequence from i to j-1.
- `toString():` - returns a string representing the data in this sequence.

---

# Collections in Java

### What are Java Collections ?

- A Collection is a group of individual objects represented as a single unit.
- Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

### Two main root interfaces of Java collection classes

- The Collection interface: **java.util.Collection**
- Map interface: **java.util.Map**

## Collections Framework

- A Java collection framework provides an architecture to store and manipulate a group of objects.
- A Java collection framework includes the following:
    - **Collection Interfaces:**
        - Interface in Java refers to the abstract data types.
        - They allow Java collections to be manipulated independently from the details of their representation.
        - Also, they form a hierarchy in object-oriented programming languages.
    - **Collection Classes:**
        - Classes in Java are the implementation of the collection interface.
        - It basically refers to the data structures that are used again and again.
    - **Collections Algorithms:**
        - Algorithm refers to the methods which are used to perform operations such as searching and sorting, on objects that implement collection interfaces.
        - Algorithms are polymorphic in nature as the same method can be used to take many forms or you can say perform different implementations of the Java collection interface.
- The Java collection framework provides the developers to access prepackaged data structures as well as algorithms to manipulate data.

## Collections: Interfaces

- **Iterator interface**
    - Iterataor is an interface that iterates the elements.

- It is used to traverse the list and modify the elements.
- Iterator interface has three methods which are mentioned below:
    a. **public boolean hasNext()** – This method returns true if iterator has more elements.
    b. **public object next()** – It returns the element and moves the cursor pointer to the next element. 3. **public void remove()** – This method removes the last elements returned by the iterator.

- **Collection Interface**

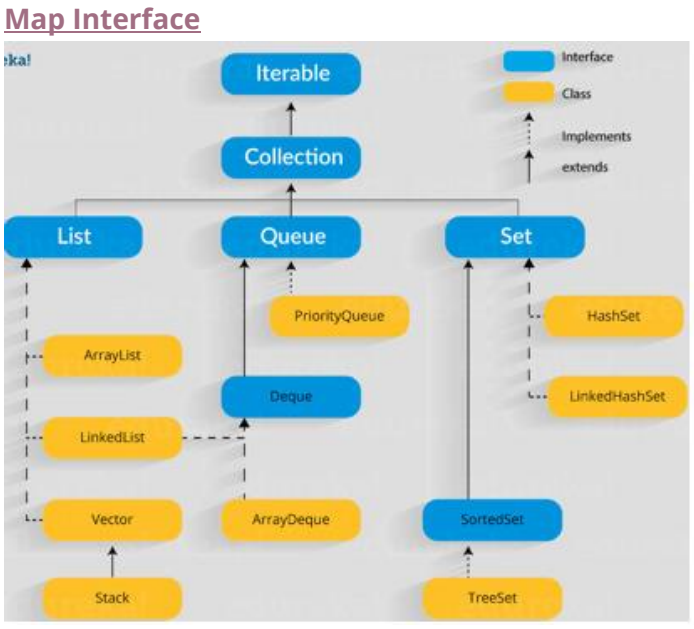
  There are **3 components that extend the collection interface:**

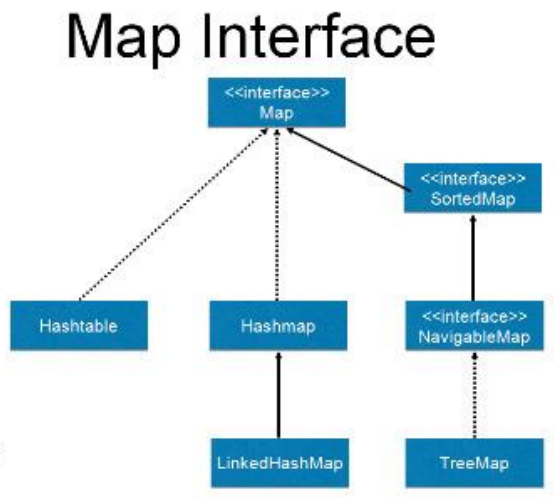
i.List

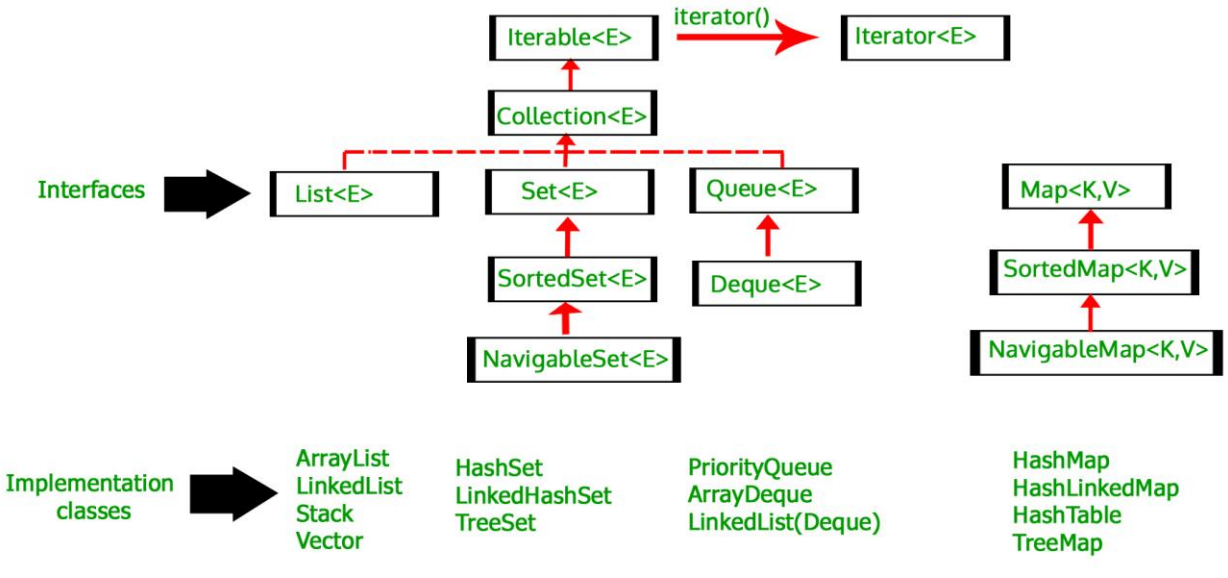
ii. Queue

iii. Sets

- **Map Interface**




Important Points:

- **Collection:**
    - Root interface with basic methods .
    - Like add(), remove(), contains(), isEmpty(), addAll(), ... etc.
- **Set:**
    - Doesn't allow duplicates.
    - Example implementations of Set interface are HashSet (Hashing based) and TreeSet (balanced BST based).
    - Note that TreeSet implements SortedSet.
- **List:**
    - Can contain duplicates and elements are ordered.
    - Example implementations are LinkedList (linked list based) and ArrayList (dynamic array based)
- **Queue:**
    - Typically order elements in FIFO order except exceptions like PriorityQueue.
- **Deque:**
    - Elements can be inserted and removed at both ends.
    - Allows both LIFO and FIFO.
- **Map:**
    - Contains Key value pairs. Doesn't allow duplicates.
    - Example implementation are HashMap and TreeMap.
    - TreeMap implements SortedMap.
- The difference between Set and Map interface is that in Set we have only keys, whereas in Map, we have key, value pairs.

### Advantages of Collection Framework:

1. **Consistent API :** The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some* common set of methods.
2. **Reduces programming effort:** A programmer doesn't have to worry about the design of Collection, and he can focus on its best use in his program.
3. **Increases program speed and quality:** Increases performance by providing high-performance implementations of useful data structures and algorithms.

# Java Collections: Lists

### What are Lists in Java ?

- A List is an ordered Collection of elements which may contain duplicates.
- It is an interface that extents the Collection interface.

Implementations:

- In order to use the List Interface, we ***need to instantiate a concrete class***.
- Following are the few implementations that can be used:
    - util.ArrayList
    - util.LinkedList
    - util.Vector
    - util.Stack
- In all the above implementations only **Vector** is **thread-safe**.
- For others we need to make it thread safe using `Collections.synchronizedList`.

```
import java.math.BigDecimal;
import java.util.*;

public class ListExample {
    public static void main(String[] args){
        List<String> arrayList = new ArrayList<>();
        List<Integer> linkedList = new LinkedList<>();
        List<Double> vectorList = new Vector<>();
        List<BigDecimal> stackList = new Stack<>();
    }
}
```

## 1. ArrayList

- ArrayList is a re-sizable array, also called a ***dynamic array*** ***similar to vector in C++***.
- It grows its size to accommodate new elements and shrinks the size when the elements are removed.
- ArrayList ***inherits AbstractList class*** and ***implements List interface***.
- It internally uses an array to store the elements. Just like arrays, It allows you to retrieve the elements by their index.
- It allows duplicate and null values.
- It is an ordered collection maintains the insertion order of the elements.
- ***ArrayList can not be used for primitive types, like*** `int`, `char`***, etc, need a wrapper class for such cases*** (refer this for details).
- We need to use boxed types like `Integer`, `Character`, `Boolean` etc.
- It is **not synchronized**, if multiple threads try to modify it at the same time, then the final outcome will be non-deterministic.
- We must explicitly synchronize access to an ArrayList if multiple threads are gonna modify it.

### Constructors in Java ArrayList:

- **ArrayList():** This constructor is used to build an empty array list.
- **ArrayList(Collection c):** This constructor is used to build an array list initialized with the elements from collection c.
- **ArrayList(int capacity):** This constructor is used to build an array list with initial capacity being specified.

### Syntax

```
List<Integer> al = new ArrayList<>();
ArrayList<Integer> arrli = new ArrayList<>();
```

## 1. Creating a ArrayList and adding new elements to it

- Use `ArrayList()` to create an empty Array List.
- Use `add(element)` and `add(index, element)` methods to add elements.
- Use `ArrayList(collection)` to create Array List from existing collection.
- Use `addAll(collection)` to add all the elements of a collection at once.

```
public class Ex1CreateArrayListAndAddElements {
```

```
    public static void main(String[] args){
        // Creation an empty ArrayList using :=> ArrayList()
        List<String> friends = new ArrayList<>();

        // Adding new elements to the ArrayList using :=> add(element)
        friends.add("Rajeev");
        friends.add("John");
        friends.add("David");
        friends.add("Chris");
        System.out.println("Initial Friend List: " + friends);

        // Add an element at a particular index in an ArrayList using :=> add(index, element)
        friends.add(2, "Steve");
        System.out.println("Friends List after friends.add(2, \"Steve\") : " + friends);

        // Create a  ArrayList from existing collection using :=> ArrayList(collection)
        ArrayList<String> familyFriends = new ArrayList<>(friends);
        System.out.println("\nFamily Friends initially : " + familyFriends);

        ArrayList<String> spouseFriends = new ArrayList<>();
        spouseFriends.add("Jesse");
        spouseFriends.add("Walt");
        System.out.println("Spouse Friends : " + spouseFriends);

        // Add a collection to existing using :=> addAll(collection)
        familyFriends.addAll(spouseFriends);
        System.out.println("Family Friends after addAll(spouseFriends) : " + familyFriends);
    }
}
```

**Output:**

```
First 10 Primes are: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

## 2. Access and Modify elements in an ArrayList

- Check if an ArrayList is empty using the `isEmpty()` method.
- Find the size of an ArrayList using the `size()` method.
- Access the element at a particular index in an ArrayList using the `get(index)` method.
- Modify the element at a particular index in an ArrayList using the `set(index, element)` method.

```
public class Ex2AccessModifyElementInArrayList {
    public static void main(String[] args){
        List<String> topCompanies = new ArrayList<>();

        // Check if an ArrayList is empty using :=>  isEmpty()
        System.out.println("Is the topCompanies list empty? : " + topCompanies.isEmpty());

        topCompanies.add("Google");
        topCompanies.add("Apple");
        topCompanies.add("Microsoft");
        topCompanies.add("Amazon");
        topCompanies.add("Facebook");

        // Find the size of the ArrayList using :=> size()
        System.out.println("Here are the top " + topCompanies.size() + " companies in the world.");
        System.out.println(topCompanies);

        // Retrieve the element at first, last and given index using :=> get(index)
        String bestCompany = topCompanies.get(0);
        String lastCompany = topCompanies.get(topCompanies.size()-1);
        String thirdBestCompany = topCompanies.get(2);

        System.out.println("Best Company: " + bestCompany);
        System.out.println("Third Best Company: " + thirdBestCompany);
        System.out.println("Last Best Company: " + lastCompany);

        // Modify the element at a given index using :=> set(index, element)
        topCompanies.set(4, "Walmart");

        System.out.println("Modified Top Companies List: " + topCompanies);
    }
}
```

**Output:**

```
Is the topCompanies list empty? : true
Here are the top 5 companies in the world.
[Google, Apple, Microsoft, Amazon, Facebook]
Best Company: Google
Third Best Company: Microsoft
Last Best Company: Facebook
Modified Top Companies List: [Google, Apple, Microsoft, Amazon, Walmart]
```

## 3. Removing elements from an ArrayList

- Remove the element at a given index in an ArrayList using `remove(index)`.
- Remove an element from an ArrayList using `remove(element)`.
- Remove all the elements from an ArrayList that exist in a given collection using `removeAll(collection)`.
- Remove all the elements matching a given predicate using `removeIf(filterCondition)`.
- Clear an ArrayList completely using `clear()`.

```
public class Ex3RemoveElementFromArrayList {
    public static void main(String[] args) {
        List<String> programmingLanguages = new ArrayList<>();
        programmingLanguages.add("C");
        programmingLanguages.add("C++");
        programmingLanguages.add("Java");
        programmingLanguages.add("Kotlin");
        programmingLanguages.add("Python");
        programmingLanguages.add("Perl");
```

```
        programmingLanguages.add("Ruby");

        System.out.println("Initial List : " + programmingLanguages);

        // Remove element at an index using :=> remove(index)
        programmingLanguages.remove(5);
        System.out.println("After remove(5) : " + programmingLanguages);

        // Remove first occurrence of element using :=> remove(element)
        // remove() method returns false if element doesn't exists
        boolean isKotlinRemoved = programmingLanguages.remove("Kotlin");
        System.out.println("After remove(\"Kotlin\") : " + programmingLanguages);

        List<String> scriptingLanguages = new ArrayList<>();
        scriptingLanguages.add("Python");
        scriptingLanguages.add("Ruby");
        scriptingLanguages.add("Perl");

        // Remove all the elements that exist in a given collection using :=> removeAll(collection)
        programmingLanguages.removeAll(scriptingLanguages);
        System.out.println("After removeAll(scriptingLanguages) : " + programmingLanguages);

        // Remove all the elements that satisfy a given condition with
        // Predicate using :=> removeIf(filterCondition)
        programmingLanguages.removeIf(s -> s.startsWith("C"));
        System.out.println("After Removing all elements that start with \"C\" : " + programmingLanguages);

        // Remove all the elements from the ArrayList using :=> clear()
        programmingLanguages.clear();
        System.out.println("After clear() : " + programmingLanguages );
    }
}
```

**Output:**

```
Initial List : [C, C++, Java, Kotlin, Python, Perl, Ruby]
After remove(5) : [C, C++, Java, Kotlin, Python, Ruby]
After remove("Kotlin") : [C, C++, Java, Python, Ruby]
After removeAll(scriptingLanguages) : [C, C++, Java]
After Removing all elements that start with "C" : [Java]
After clear() : []
```

## 4. Iterating over an ArrayList

1. Simple for loop with index `for(index, condition, inc/dec)`.
2. Enhanced for loop `for (element : collection)`.
3. Java 8 `forEach(e -> action on e)` and lambda expression.
4. `iterator()` and **hasNext()**.
5. `iterator()` and Java 8 `forEachRemaining(e -> action on e)` method.
6. `listIterator()` and **hasPrevious()** and **haNext()**.
7. The `iterator()` and `listIterator()` methods are useful when we need to modify the ArrayList while traversing.
8. We can remove elements from the ArrayList using `iterator.remove()` method while traversing through it.

```
public class Ex4IterateOverArrayList {
    public static void main(String[] args){
        List<String> tvShows = new ArrayList<>();
        tvShows.add("Breaking Bad");
        tvShows.add("Game of Thrones");
        tvShows.add("Friends");
        tvShows.add("Prison Break");

        // Iterate by simple for loop with index using :=> for(index, condition, inc/dec)
        System.out.println("\n=== Iterate using simple for loop with index ===");
        for (int i = 0; i < tvShows.size(); ++i){
            System.out.println(tvShows.get(i));
        }

        // Iterate by enhanced for loop using :=> for (element : collection)
        System.out.println("\n=== Iterate using enhanced for loop ===");
        for(String tvShow : tvShows){
            System.out.println(tvShow);
        }

        // Iterate by Java8 forEach and lambda using :=> forEach(e -> { action using e })
        System.out.println("\n=== Iterate using Java 8 forEach and lambda ===");
        tvShows.forEach(tvShow -> {
            System.out.println(tvShow);
        });

        // Iterate by Iterator using :=> iterator() and hasNext()
        System.out.println("\n=== Iterate using an iterator() ===");
        Iterator<String> tvShowIterator = tvShows.iterator();
        while (tvShowIterator.hasNext()){
            String tvShow = tvShowIterator.next();
            System.out.println(tvShow);
        }

        // Iterate by Iterator and Java 8 forEachRemaining using
        // :=> iterator() and forEachRemaining(e -> { action using e }
        System.out.println("\n=== Iterate using an iterator() and Java 8 forEachRemaining() method ===");
        tvShowIterator = tvShows.iterator();
        tvShowIterator.forEachRemaining(tvShow -> {
            System.out.println(tvShow);
        });

        // Iterate by ListIterator to go in both directions using
        // :=> listIterator(), hasPrevious() and hasNext()
        System.out.println("\n=== Iterate using a listIterator() to traverse in both directions ===");
        ListIterator<String> tvShowListIterator = tvShows.listIterator(tvShows.size()); // here starts from end
        while (tvShowListIterator.hasPrevious()){
            String tvShow = tvShowListIterator.previous();
```

```
            System.out.println(tvShow);
        }

        // Modify list while traversing by removing elements using :=> iterator.remove()
        tvShowIterator = tvShows.iterator();
        while (tvShowIterator.hasNext()){
            String movieName = tvShowIterator.next();
            if (movieName.startsWith("Game") || movieName.endsWith("Bad")){
                tvShowIterator.remove();
            }
        }
        System.out.println("\n=== Final Movies List after iterator.remove() : ===\n" + tvShows);
    }
}
```

**Output:**

```
=== Iterate using simple for loop with index ===
Breaking Bad
Game of Thrones
Friends
Prison Break

=== Iterate using enhanced for loop ===
Breaking Bad
Game of Thrones
Friends
Prison Break

=== Iterate using Java 8 forEach and lambda ===
Breaking Bad
Game of Thrones
Friends
Prison Break

=== Iterate using an iterator() ===
Breaking Bad
Game of Thrones
Friends
Prison Break

=== Iterate using an iterator() and Java 8 forEachRemaining() method ===
Breaking Bad
Game of Thrones
Friends
Prison Break

=== Iterate using a listIterator() to traverse in both directions ===
Prison Break
Friends
Game of Thrones
Breaking Bad

=== Final Movies List after iterator.remove() : ===
[Friends, Prison Break]
```

## 5. Searching for elements in an ArrayList

- Check if an ArrayList contains a given element `contains(element)`.
- Index of the first occurrence of an element in an ArrayList `indexOf(element)`.
- Index of the last occurrence of an element in an ArrayList `lastIndexOf(element)`.

```
public class Ex5SearchInArrayList {
    public static void main(String[] args){
        List<String> names = new ArrayList<>();
        names.add("John");
        names.add("Alice");
        names.add("Bob");
        names.add("Steve");
        names.add("John");
        names.add("Steve");
        names.add("Maria");

        // Check if an ArrayList contains a given element using :=> contains(element)
        System.out.println("Does name array contains \"Bob\" ? : " + names.contains("Bob"));

        // Find the index of the first occurrence of an element in an ArrayList using :=> indexOf(element)
        System.out.println("Index of \"Steve\" : " + names.indexOf("Steve"));
        System.out.println("Index of \"Mark\" : " + names.indexOf("Mark"));

        // Find the index of the last occurrence of an element in an ArrayList
        System.out.println("Last Index of \"John\" : " + names.lastIndexOf("John"));
        System.out.println("Last Index of \"Bill\" : " + names.lastIndexOf("Bill"));
    }
}
```

**Output:**

```
Does name array contains "Bob" ? : true
Index of "Steve" : 3
Index of "Mark" : -1
Last Index of "John" : 4
Last Index of "Bill" : -1
```

## 6. Sorting an ArrayList using Collections.sort()

- We can sort any collection in Java using `Collections.sort(collection, lambda_comparator)`.

```
public class Ex6SortArrayListUsingCollectionsSort {
    public static void main(String[] args){
        ArrayList<Integer> numbers = new ArrayList<>();
```

```
        numbers.add(13);
        numbers.add(7);
        numbers.add(18);
        numbers.add(5);
        numbers.add(2);

        System.out.println("Numbers Initially: " + numbers);

        // Sorting an ArrayList using :=> Collections.sort(collection)
        Collections.sort(numbers);

        System.out.println("Numbers in Natural Order (Sorted Order) : " + numbers);


        //=========== Creating and Sorting User Defined DataTypes ========/
        List<User> users = new ArrayList<>();
        users.add(new User("Sachin", 47));
        users.add(new User("Chris", 34));
        users.add(new User("Rajeev", 25));
        users.add(new User("David", 31));
        users.add(new User("Chris", 54));

        System.out.println("\nUsers List: " + users);
        // Sort users by age
        Collections.sort(users, (user1, user2) -> user1.getAge() - user2.getAge());
        System.out.println("Users List sorted by age: " + users);

        // Sort users by name
        Collections.sort(users, (user1, user2) -> user1.getName().compareTo(user2.getName()));
        System.out.println("Users List sorted by name: " + users);

        // Sort users by name if same name, reverse sort using age
        Collections.sort(users, (user1, user2) -> {
            int val = user1.getName().compareTo(user2.getName());
            return val != 0 ? val : user2.getAge() - user1.getAge();
        });
        System.out.println("Users sorted by name and for same name reverse sorted by age: " + users);

    }
}
```

**Output:**

```
Numbers Initially: [13, 7, 18, 5, 2]
Numbers in Natural Order (Sorted Order) : [2, 5, 7, 13, 18]

Users List: [User{name='Sachin', age=47}, User{name='Chris', age=34}, User{name='Rajeev', age=25}, User{name='David',
age=31}, User{name='Chris', age=54}]
Users List sorted by age: [User{name='Rajeev', age=25}, User{name='David', age=31}, User{name='Chris', age=34},
User{name='Sachin', age=47}, User{name='Chris', age=54}]
Users List sorted by name: [User{name='Chris', age=34}, User{name='Chris', age=54}, User{name='David', age=31},
User{name='Rajeev', age=25}, User{name='Sachin', age=47}]
Users sorted by name and for same name reverse sorted by age: [User{name='Chris', age=54}, User{name='Chris', age=34},
User{name='David', age=31}, User{name='Rajeev', age=25}, User{name='Sachin', age=47}]
```

## 7. Sorting an ArrayList using ArrayList.sort() method

- We can also sort an ArrayList using `arrayList.sort(lambda_comparator)`.

```
public class Ex7SortArrayListUsingArrayListSort {
    public static void main(String[] args){
        List<String> names = new ArrayList<>();
        names.add("Lisa");
        names.add("Jennifer");
        names.add("Mark");
        names.add("David");

        System.out.println("Names: " + names);

        // Sorting an ArrayList using :=> sort(lambdaExpression)
        names.sort((name1, name2) -> name1.compareTo(name2));
        // A concise way using :=> sort(Comparator)
        names.sort(Comparator.naturalOrder());

        System.out.println("Sorted Names: " + names);


        //=========== Creating and Sorting User Defined DataTypes ========/
        List<User> users = new ArrayList<>();
        users.add(new User("Sachin", 47));
        users.add(new User("Chris", 34));
        users.add(new User("Rajeev", 25));
        users.add(new User("David", 31));
        users.add(new User("Chris", 54));

        System.out.println("\nUsers List: " + users);
        // Sort users by age
        users.sort((user1, user2) -> user1.getAge() - user2.getAge());
        System.out.println("Users List sorted by age: " + users);

        // Sort users by name
        users.sort((user1, user2) -> user1.getName().compareTo(user2.getName()));
        System.out.println("Users List sorted by name: " + users);

        // Sort users by name if same name, reverse sort using age
        users.sort((user1, user2) -> {
            int val = user1.getName().compareTo(user2.getName());
            return val != 0 ? val : user2.getAge() - user1.getAge();
        });
        System.out.println("Users sorted by name and for same name reverse sorted by age: " + users);
    }
}
```

**Output:**

```
Names: [Lisa, Jennifer, Mark, David]
Sorted Names: [David, Jennifer, Lisa, Mark]

Users List: [User{name='Sachin', age=47}, User{name='Chris', age=34}, User{name='Rajeev', age=25}, User{name='David', age=31}, User{name='Chris', age=54}]
Users List sorted by age: [User{name='Rajeev', age=25}, User{name='David', age=31}, User{name='Chris', age=34}, User{name='Sachin', age=47}, User{name='Chris', age=54}]
Users List sorted by name: [User{name='Chris', age=34}, User{name='Chris', age=54}, User{name='David', age=31}, User{name='Rajeev', age=25}, User{name='Sachin', age=47}]
Users sorted by name and for same name reverse sorted by age: [User{name='Chris', age=54}, User{name='Chris', age=34}, User{name='David', age=31}, User{name='Rajeev', age=25}, User{name='Sachin', age=47}]
```

## 8. Thread Unsafe ArrayList when Multiple Threads modifies concurrently

- The ArrayList class is not synchronized.
- If multiple threads try to modify an ArrayList at the same time then the final result becomes not-deterministic because one thread might override the changes done by another thread.

```
public class Ex8ThreadUnsafeArrayList {
    public static void main(String[] args) throws InterruptedException {
        List<Integer> unsafeArrayList = new ArrayList<>();
        unsafeArrayList.add(1);
        unsafeArrayList.add(2);
        unsafeArrayList.add(3);

        // Create a thread pool of size 10
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // Create a runnable task that increments each element of ArrayList by 1
        Runnable task = () -> incrementArrayList(unsafeArrayList);

        // Submit the task to the executor service 100 times.
        // All the tasks will modify the ArrayList concurrently
        for (int i = 0; i < 100; ++i){
            executorService.submit(task);
        }

        // Shutdown the executor and wait for termination
        executorService.shutdown();
        executorService.awaitTermination(60, TimeUnit.SECONDS);

        System.out.println(unsafeArrayList);
    }

    private static void incrementArrayList(List<Integer> unsafeArrayList) {
        for (int i = 0; i < unsafeArrayList.size(); ++i) {
            unsafeArrayList.set(i, unsafeArrayList.get(i) + 1);
        }
    }
}
```

**Output:**

```
[93, 96, 94]
```

**Note:** The final output of the above program should be equal to `[101, 102, 103]` because we're incrementing the values in the ArrayList 100 times. But if we run the program, it will produce different output every time it is run.

## 9. Making ArrayList Thread Safe by Synchronizing Access to an ArrayList

- We can use `Collections.synchronizedList()` method to get a synchronized view of the ArrayList.
- While doing modifications to the ArrayList inside the `incrementArrayList()` method we will wrap it inside a `synchronized(safeArrayList){}` block.
- This ensures that no two threads can increment ArrayList elements at the same time.
- We can also use a `CopyOnWriteArrayList` if you need thread safety.
- It is a thread-safe version of the ArrayList class. It implements all the mutating operations by making a fresh copy of the ArrayList.
- Unlike the previous program, the output of this program is deterministic and will always be the same.

```
public class Ex9ThreadSafeArrayList {
    public static void main(String[] args) throws InterruptedException {
        // Creating Thread Safe ArrayList
        List<Integer> safeArrayList = Collections.synchronizedList(new ArrayList<>());
        safeArrayList.add(1);
        safeArrayList.add(2);
        safeArrayList.add(3);

        // Create a thread pool of size 10
        ExecutorService executorService = Executors.newFixedThreadPool(10);

        // Create a Runnable task that increments each element of the ArrayList by one
        Runnable task = () -> incrementArrayList(safeArrayList);

        // Submit the task to the executor service 100 times
        // All the will modify ArrayList concurrently
        for (int i = 0; i < 100; ++i) {
            executorService.submit(task);
        }

        // Shutdown the executor and wait for termination
        executorService.shutdown();
        executorService.awaitTermination(60, TimeUnit.SECONDS);

        System.out.println(safeArrayList);
```

```
    }

    private static void incrementArrayList(List<Integer> safeArrayList){
        synchronized (safeArrayList){ // ensures no two threads performs on this code block at the same time
            for(int i = 0; i < safeArrayList.size(); ++i){
                safeArrayList.set(i, safeArrayList.get(i)+1);
            }
        }
    }
}
```

**Output:**

```
[101, 102, 103]
```

# 2. Linked List

- LinkedList is a **doubly linked list** implementation of Java's `List` and `Deque` interfaces.
- LinkedList maintains the insertion order of the elements and can have duplicate and null values.
- Implements `Queue` and `Deque` interfaces, and hence can also be used as a `Queue`, `Deque` or `Stack`.
- **Not thread-safe**, must explicitly synchronize concurrent modifications to the LinkedList in a multi-threaded environment.

### Syntax:

```
Linkedlist<Integer> linkLi = new Linkedlist<>();
```

### Java ArrayList vs LinkedList

- **Implementation:** Both implement the `List` interface, but differ completely in the way they store and link to the elements.
- **Storing:** ArrayList stores the elements sequentially based on their index, but LinkedList uses a doubly-linked list to store its elements.
- **Memory Cosumption:** LinkedList consumes more memory because it also stores the next and previous references along with the data.
- **Element Access:** We can access an element in an ArrayList in **O(1)** time, but it takes **O(n)** time to access an element in a LinkedList
- **List Modification:** Adding or removing elements are usually slower in an ArrayList compared to LinkedList due to shifting and resizing.

## 1. Creating a LinkedList and adding new elements to it

- Use `LinkedList()` to create an empty Linked List.
- Use `add(element)` and `add(index, element)` methods to add elements.
- Also use `addFirst()` and `addLast()` method to add elements, these methods come from `Deque` interface.
- Use `LinkedList(collection)` to create Linked List with existing collection.
- Use `addAll(collection)` to add all the elements of a collection at once.

```
public class Ex1CreateLinkedListAndAddElements {
    public static void main(String[] args){
        // Creation an empty LinkedList using :=> LinkedList()
        LinkedList<String> friends = new LinkedList<>();

        // Adding elements using :=> add(element)
        friends.add("Rajeev");
        friends.add("John");
        friends.add("David");
        friends.add("Chris");
        System.out.println("Initial Friend List: " + friends);

        // Add elements at a specific index using :=> add(index, element)
        friends.add(3, "Lisa");
        System.out.println("Friends List after add(3, \"Lisa\") : " + friends);

        // Add an element at the beginning of Linked List :=> addFirst(element)
        friends.addFirst("Steve");
        System.out.println("Friends List after addFirst(\"Steve\") : " + friends);

        // Add an element at end (equivalent to add method) using :=> addLast(element)
        friends.addLast("Jennifer");
        System.out.println("Friends List after addLast(\"Jennifer\") : " + friends);

        // Create a Linked List from existing collection using :=> LinkedList(collection)
        LinkedList<String> familyFriends = new LinkedList<>(friends);
        System.out.println("\nFamily Friends initially : " + familyFriends);

        LinkedList<String> spouseFriends = new LinkedList<>();
        spouseFriends.add("Jesse");
        spouseFriends.add("Walt");
        System.out.println("Spouse Friends : " + spouseFriends);

        // Add a collection to existing using :=> addAll(collection)
        familyFriends.addAll(spouseFriends);
        System.out.println("Family Friends after addAll(spouseFriends) : " + familyFriends);
    }
}
```

**Output:**

```
Initial Friend List: [Rajeev, John, David, Chris]
Friends List after add(3, "Lisa") : [Rajeev, John, David, Lisa, Chris]
Friends List after addFirst("Steve") : [Steve, Rajeev, John, David, Lisa, Chris]
Friends List after addLast("Jennifer") : [Steve, Rajeev, John, David, Lisa, Chris, Jennifer]
```

```
Family Friends initially : [Steve, Rajeev, John, David, Lisa, Chris, Jennifer]
Spouse Friends : [Jesse, Walt]
Family Friends after addAll(spouseFriends) : [Steve, Rajeev, John, David, Lisa, Chris, Jennifer, Jesse, Walt]
```

## 2. Access and Modify elements in a LinkedList

- Get the first element in the LinkedList using `getFirst()` and last element using `getLast()`.
- Both the above methods gives **NoSuchElementException** if the LinkedList is empty.
- Get the element at a given position using `get(index)`.
- Change the element at a particular index using `set(index, val)`.

```java
public class Ex2AccessModifyElementsInLinkedList {
    public static void main(String[] args){
        LinkedList<String> topCompanies = new LinkedList<>();

        // Check if an LinkedList is empty using :=>  isEmpty()
        System.out.println("Is the topCompanies list empty? : " + topCompanies.isEmpty());

        topCompanies.add("Google");
        topCompanies.add("Apple");
        topCompanies.add("Microsoft");
        topCompanies.add("Amazon");
        topCompanies.add("Facebook");

        // Find the size of the LinkedList using :=> size()
        System.out.println("Here are the top " + topCompanies.size() + " companies in the world.");
        System.out.println(topCompanies);

        // Retrieve the element at first, last and given index using :=> get(index)
        String bestCompany = topCompanies.getFirst();
        String lastCompany = topCompanies.getLast();
        String thirdBestCompany = topCompanies.get(2);

        System.out.println("Best Company: " + bestCompany);
        System.out.println("Third Best Company: " + thirdBestCompany);
        System.out.println("Last Best Company: " + lastCompany);

        // Modify the element at a given index using :=> set(index, element)
        topCompanies.set(4, "Walmart");

        System.out.println("Modified Top Companies List: " + topCompanies);
    }
}
```

**Output:**

```
Is the topCompanies list empty? : true
Here are the top 5 companies in the world.
[Google, Apple, Microsoft, Amazon, Facebook]
Best Company: Google
Third Best Company: Microsoft
Last Best Company: Facebook
Modified Top Companies List: [Google, Apple, Microsoft, Amazon, Walmart]
```

## 3. Removing elements from an LinkedList

- Remove last element using `remove()` or `removeLast()` and first element using `removeFirst()` .
- Above methods gives **NoSuchElementException** if the LinkedList is empty.
- Remove element at an index using `remove(index)`.
- Remove first occurrence of element using `remove(element)`.
- Remove all the elements that exist in a given collection using `removeAll(collection)`.
- Remove all the elements that satisfy a given condition with Predicate using `removeIf(filterCondition)`.
- Remove all the elements from the ArrayList using `clear()`.

**Note: As Queue, Deque and Stack can also be implemented using LinkedList, so it additionally gives methods suited to them.**

- ***Add*** element at last using `offer(element)` or `offerLast(element)` and at start using `offerFirst(element)`.
- ***Retrieve*** element without removing using methods `peek()`, `peekFirst()` and `peekLast()`.
- ***Remove and retrieve*** using methods `poll()`, `pollFirst()` and `pollLast()`.
- For stack represented using LinkedList `push()` to add element and `pop()` to remove.

```java
public class Ex3RemoveElementFromLinkedList {
    public static void main(String[] args){
        LinkedList<String> programmingLanguages = new LinkedList<>();
        programmingLanguages.add("Fortran");
        programmingLanguages.add("C");
        programmingLanguages.add("C++");
        programmingLanguages.add("Java");
        programmingLanguages.add("Kotlin");
        programmingLanguages.add("Python");
        programmingLanguages.add("Perl");
        programmingLanguages.add("Ruby");
        programmingLanguages.add("GoLang");

        System.out.println("Initial LinkedList : " + programmingLanguages);

        // Remove first element using :=> removeFirst()
        programmingLanguages.removeFirst(); // Throws NoSuchElementException if empty
        System.out.println("After removeFirst() : " + programmingLanguages);

        // Remove last element using :=> removeLast()
        programmingLanguages.removeLast(); // Throws NoSuchElementException if empty
```

```
        System.out.println("After removeLast() : " + programmingLanguages);

        // Remove element at an index using :=> remove(index)
        programmingLanguages.remove(5);
        System.out.println("After remove(5) : " + programmingLanguages);

        // Remove first occurrence of element using :=> remove(element)
        // remove() method returns false if element doesn't exists
        boolean isKotlinRemoved = programmingLanguages.remove("Kotlin");
        System.out.println("After remove(\"Kotlin\") : " + programmingLanguages);

        LinkedList<String> scriptingLanguages = new LinkedList<>();
        scriptingLanguages.add("Python");
        scriptingLanguages.add("Ruby");
        scriptingLanguages.add("Perl");

        // Remove all the elements that exist in a given collection using :=> removeAll(collection)
        programmingLanguages.removeAll(scriptingLanguages);
        System.out.println("After removeAll(scriptingLanguages) : " + programmingLanguages);

        // Remove all the elements that satisfy a given condition with
        // Predicate using :=> removeIf(filterCondition)
        programmingLanguages.removeIf(s -> s.startsWith("C"));
        System.out.println("After Removing all elements that start with \"C\" : " + programmingLanguages);

        // Remove all the elements from the LinkedList using :=> clear()
        programmingLanguages.clear();
        System.out.println("After clear() : " + programmingLanguages );
    }
}
```

**Output:**

```
Initial LinkedList : [Fortran, C, C++, Java, Kotlin, Python, Perl, Ruby, GoLang]
After removeFirst() : [C, C++, Java, Kotlin, Python, Perl, Ruby, GoLang]
After removeLast() : [C, C++, Java, Kotlin, Python, Perl, Ruby]
After remove(5) : [C, C++, Java, Kotlin, Python, Ruby]
After remove("Kotlin") : [C, C++, Java, Python, Ruby]
After removeAll(scriptingLanguages) : [C, C++, Java]
After Removing all elements that start with "C" : [Java]
After clear() : []
```

## 4. Iterating Over a Linked List

- Enhanced for loop `for (element : collection)`.
- Java 8 `forEach(e -> action on e)` and lambda expression.
- `iterator()` and **hasNext()**.
- `iterator()` and Java 8 `forEachRemaining(e -> action on e)` method.
- `descedingIterator` and **hasNext()** to iterate from end.
- `listIterator()` and **hasPrevious()** and **haNext()**.
- The `iterator()` and `listIterator()` methods are useful when we need to modify the LinkedList while traversing.
- We can remove elements from the ArrayList using `iterator.remove()` method while traversing through it.

```
public class Ex4IterateOverLinkedList {
    public static void main(String[] args){
        LinkedList<String> tvShows = new LinkedList<>();
        tvShows.add("Breaking Bad");
        tvShows.add("Game of Thrones");
        tvShows.add("Friends");
        tvShows.add("Prison Break");

        // Iterate by enhanced for loop using :=> for (element : collection)
        System.out.println("\n=== Iterate using enhanced for loop ===");
        for(String tvShow : tvShows){
            System.out.println(tvShow);
        }

        // Iterate by Java8 forEach and lambda using :=> forEach(e -> { action using e })
        System.out.println("\n=== Iterate using Java 8 forEach and lambda ===");
        tvShows.forEach(tvShow -> {
            System.out.println(tvShow);
        });

        // Iterate by Iterator using :=> iterator() and hasNext()
        System.out.println("\n=== Iterate using an iterator() ===");
        Iterator<String> tvShowIterator = tvShows.iterator();
        while (tvShowIterator.hasNext()){
            String tvShow = tvShowIterator.next();
            System.out.println(tvShow);
        }

        // Iterate by Iterator and Java 8 forEachRemaining using
        // :=> iterator() and forEachRemaining(e -> { action using e }
        System.out.println("\n=== Iterate using an iterator() and Java 8 forEachRemaining() method ===");
        tvShowIterator = tvShows.iterator();
        tvShowIterator.forEachRemaining(tvShow -> {
            System.out.println(tvShow);
        });

        // Iterate by descendingIterator using :=> descendingIterator() and hasNext()
        System.out.println("\n=== Iterate over a LinkedList using descendingIterator() ===");
        Iterator<String> tvShowDecendingIterator = tvShows.descendingIterator();
        while (tvShowDecendingIterator.hasNext()){
            String tvShow = tvShowDecendingIterator.next();
            System.out.println(tvShow);
        }

        // Iterate by ListIterator to go in both directions using
        // :=> listIterator(), hasPrevious() and hasNext()
        System.out.println("\n=== Iterate using a listIterator() to traverse in both directions ===");
        ListIterator<String> tvShowListIterator = tvShows.listIterator(tvShows.size()); // here starts from end
```

```
        while (tvShowListIterator.hasPrevious()){
            String tvShow = tvShowListIterator.previous();
            System.out.println(tvShow);
        }

        // Modify list while traversing by removing elements using :=> iterator.remove()
        tvShowIterator = tvShows.iterator();
        while (tvShowIterator.hasNext()){
            String movieName = tvShowIterator.next();
            if (movieName.startsWith("Game") || movieName.endsWith("Bad")){
                tvShowIterator.remove();
            }
        }
        System.out.println("\n=== Final Movies List after iterator.remove() : ===\n" + tvShows);
    }
}
```

**Output:**

```
=== Iterate using enhanced for loop ===
Breaking Bad
Game of Thrones
Friends
Prison Break

=== Iterate using Java 8 forEach and lambda ===
Breaking Bad
Game of Thrones
Friends
Prison Break

=== Iterate using an iterator() ===
Breaking Bad
Game of Thrones
Friends
Prison Break

=== Iterate using an iterator() and Java 8 forEachRemaining() method ===
Breaking Bad
Game of Thrones
Friends
Prison Break

=== Iterate over a LinkedList using descendingIterator() ===
Prison Break
Friends
Game of Thrones
Breaking Bad

=== Iterate using a listIterator() to traverse in both directions ===
Prison Break
Friends
Game of Thrones
Breaking Bad

=== Final Movies List after iterator.remove() : ===
[Friends, Prison Break]
```

## 5. Searching for elements in an LinkedList

- Similar to searching done in ArrayList.
- Check if an LinkedList contains a given element `contains(element)`.
- Index of the first occurrence of an element in an LinkedList `indexOf(element)`.
- Index of the last occurrence of an element in an LinkedList `lastIndexOf(element)`.

```
public class Ex5SearchInLinkedList {
    public static void main(String[] args){
        LinkedList<String> names = new LinkedList<>();
        names.add("John");
        names.add("Alice");
        names.add("Bob");
        names.add("Steve");
        names.add("John");
        names.add("Steve");
        names.add("Maria");

        // Check if an LinkedList contains a given element using :=> contains(element)
        System.out.println("Does name array contains \"Bob\" ? : " + names.contains("Bob"));

        // Find the index of the first occurrence of an element in an LinkedList using :=> indexOf(element)
        System.out.println("Index of \"Steve\" : " + names.indexOf("Steve"));
        System.out.println("Index of \"Mark\" : " + names.indexOf("Mark"));

        // Find the index of the last occurrence of an element in an LinkedList
        System.out.println("Last Index of \"John\" : " + names.lastIndexOf("John"));
        System.out.println("Last Index of \"Bill\" : " + names.lastIndexOf("Bill"));
    }
}
```

**Output:**

```
Does name array contains "Bob" ? : true
Index of "Steve" : 3
Index of "Mark" : -1
Last Index of "John" : 4
Last Index of "Bill" : -1
```

## 6. Sorting in LinkedList

- We can sort the Linked List the same way we sorted the ArrayList.
- Here also we can use `Collections.sort(list, lambda_comparator)` and `LinkedList.sort(lambda_comparator)`.

## 7. Making LinkedList Thread Safe (Synchronization)

- We can synchronize the LinkedList the same way as we did for ArrayList.

```
List<String> safeLinkedList = Collections.synchronizedList(new LinkedList<>());
```

# 3. Vector

- It implements a growable array of objects.
- Vectors basically fall in legacy classes but now it is fully compatible with collections.
- Vector implements a dynamic array that means it can grow or shrink as required.
- Like an array, it contains components that can be ***accessed using an integer index***.
- It is very similar to ArrayList but **Vector is synchronised** .
- It extends **AbstractList** and implements **List** interfaces.

# 4. Stack

- Java Collection framework provides a Stack class which models and implements Stack data structure.
- The class is based on the basic principle of last-in-first-out.
- With `push(element)` and `pop()` operations, the class provides three more functions of `empty()`, `search(element)` and `peek()`.
- `search(element):` Searches for element in the stack, returns offset from the top of the stack if found else returns -1.
- The class can also be said to **extend Vector** and ***treats the class as a stack with the five mentioned functions***.
- The class can also be referred to as the subclass of Vector.

```
public class ExStackOperations {
    public static void main(String[] args){
        Stack<String> topCompanies = new Stack<>();

        // Check if an Stack is empty using :=>  isEmpty()
        System.out.println("Is the topCcompanies stack empty? : " + topCompanies.isEmpty());

        // Push the element to stack using :=> push(element)
        topCompanies.push("Facebook");
        topCompanies.push("Amazon");
        topCompanies.push("Microsoft");
        topCompanies.push("Apple");
        topCompanies.push("Google");

        // Find the size of the Stack using :=> size()
        System.out.println("Here are the top " + topCompanies.size() + " companies in the world :");
        System.out.println(topCompanies);

        // Retrieve and remove the top element using :=> pop()
        String bestCompany = topCompanies.pop();

        // Only retrieve the top element now using :=> top()
        String secondBestCompany = topCompanies.peek();

        System.out.println("Best Company: " + bestCompany);
        System.out.println("Second Best Company: " + secondBestCompany);

        // Search the element in stack using :=> search(element)
        System.out.println("Find \"Amazon\" : " + topCompanies.search("Amazon"));
    }
}
```

**Output:**

```
Is the topCompanies stack empty? : true
Here are the top 5 companies in the world :
[Facebook, Amazon, Microsoft, Apple, Google]
Best Company: Google
Second Best Company: Apple
Find "Amazon" : 3
```

**Note:** A more complete and consistent set of LIFO stack operations is provided by the **Deque** interface and its implementations, which should be used in preference to this class. For example:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

# Java Collections: Queues

### What are Queues in Java ?

- A queue is a data structure which follows the principle of FIFO (First-In-First-Out).
- This interface is available in the java.util.package and extends the Collection Interface.
- **LinkedList**, **PriorityQueue**, **ArrayDeque** and **ArrayBlockingQueue** are the most frequently used implementations.
- BlockingQueues have **thread-safe** implementations.
- The Queues which are available in java.util packasge are ***Unbounded Queues***.
- The Queues which are available in java.util.concurrent package are the ***Bounded Queues***.
- All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively.
- The Deques support element insertion and removal at both ends.

### Implementation:

- In order to use the queue interface, we ***need to instantiate a concrete class***.
- Following are the few implementations that can be used:
    - **Queue Interface:**
        - PriorityQueue
        - LinkedList
        - **Deque Interface:**
            - ArrayDeque
            - LinkedList
- Above implementations are **not thread safe**, **PriorityBlockingQueue** acts as an alternative for thread safe implementation.

```
import java.util.ArrayDeque;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args){
        Queue<String> linkedListQueue = new LinkedList<>();
        Queue<Integer> priorityQueueQueue = new PriorityQueue<>();
        Queue<Double> arrayDequeQueue = new ArrayDeque<>();
    }
}
```

### Methods in Queue:

| | Throws Exception | Returns Special Value |
|---|---|---|
| Insert | Add(e) | Offer(e) |
| Remove | Remove() | Poll() |
| Examine | Element() | Peek() |

### Operations on Queue:

- **Searching**, **Sorting** and **Iterating** over the queue, can be done the same way it is done for their concrete class.
- If we use **LinkedList** implementation, we can do things similar to LinkedList explained earlier.

# 1. Priority Queue / Heap

- Priority queue implements Queue interface.
- Minimum element will always be on top or head according to natural order if compartor is not provided.
- Use `PriorityQueue()` or `PriorityQueue(comaprator)` to create a heap / priority queue.
- Use `isEmpty()` to check if it is empty or `size()` to get the size.
- Use `add(element)` or `offer(element)` to add element to priority queue.
- Use `peek()` or `element()` to get the head element without removing.
- Use `remove()` or `poll()` to remove the head element.

## 1. Create simple Priority Queue / Heap

```
public class Ex1CreatePriorityQueueNormal {
    public static void main(String[] args){
        // Create a Priority Queue using :=> PriorityQueue() -> Natural Order maintained
        PriorityQueue<String> names = new PriorityQueue<>();

        // Check if Priority Queue is empty using :=> isEmpty()
        System.out.println("Is names empty ? : " + names.isEmpty());

        // Add element using :=> add(element) or offer(element)
        names.add("Rachit");
        names.add("Amit");
        names.offer("Zayed");
        names.offer("Musa");

        // Get the size of the Priority Queue using :=> size()
        System.out.println("Total names : " + names.size());

        // Get the head element of Priority Queue using :=> peek() or element()
        System.out.println("Head : "+ names.peek());
        System.out.println("Head : " + names.element());

        // Remove the element from Priority Queue using :=> remove() or poll()
        System.out.print("Removing elements one by one :");
        while (!names.isEmpty()){
            System.out.print(" " + names.remove());
        }
        System.out.println();
    }
}
```

**Output:**

```
Is names empty ? : true
Total names : 4
Head : Amit
Head : Amit
Removing elements one by one : Amit Musa Rachit Zayed
```

## 2. Create Priority Queue / Heap of user defined objects

- Priority queue needs to compare its elements and order them accordingly.
- The requirement for a PriorityQueue of user defined objects is that:
    i. Either the class ***should implement the Comparable interface and provide the implementation for the compareTo()*** function.
    ii. Or we should ***provide a custom Comparator*** while creating the PriorityQueue.

```
import java.util.Objects;
import java.util.PriorityQueue;

public class Employee implements Comparable<Employee>{
    private String name;
    private double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Employee employee = (Employee) o;
        return Double.compare(employee.salary, salary) == 0 &&
                Objects.equals(name, employee.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, salary);
    }

    @Override
    public String toString() {
        return "Employee{" +
                "name='" + name + '\'' +
                ", salary=" + salary +
                '}';
    }

    // Compare two employee objects by their salary
    @Override
    public int compareTo(Employee employee) {
        if (this.getSalary() == employee.getSalary()){
            return 0;
        } else if (this.getSalary() > employee.getSalary()) {
            return 1;
        } else {
            return -1;
        }
    }
}


public class Ex2CreatePriorityQueueUserDefinedTypes {
    public static void main(String[] args){
         // The requirement for a PriorityQueue of user defined objects is that
         //       1. Either the class should implement the Comparable interface and provide the
         //          implementation for the compareTo() function.
         //       2. Or we should provide a custom Comparator while creating the PriorityQueue.

        // Create a PriorityQueue
        PriorityQueue<Employee> employeePriorityQueue = new PriorityQueue<>();

        // Add items to the Priority Queue
        employeePriorityQueue.add(new Employee("Rajeev", 100000.00));
        employeePriorityQueue.add(new Employee("Chris", 145000.00));
        employeePriorityQueue.add(new Employee("Andrea", 115000.00));
        employeePriorityQueue.add(new Employee("Jack", 167000.00));

        // Note:- The compareTo() method implemented in the Employee class is used to determine
        //        in what order the objects should be dequeued.
        while (!employeePriorityQueue.isEmpty()) {
            System.out.println(employeePriorityQueue.remove());
        }
    }
}
```

### Output:

```
Employee{name='Rajeev', salary=100000.0}
Employee{name='Andrea', salary=115000.0}
Employee{name='Chris', salary=145000.0}
Employee{name='Jack', salary=167000.0}
```

## Deque Interface:

- The Deque is related to the ***double-ended queue*** that supports addition or removal of elements from either end of the data structure.
- It can be used as a **Queue** or as a **Stack**.
- ***Faster than Stack and LinkedList.***

### Implementations:

- Implementations of Deque:
    - ArrayDeque
    - LinkedList

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.LinkedList;

public class DequeExample {
    public static void main(String[] args){
        Deque<Integer> arrayDeque = new ArrayDeque<>();
        Deque<String> linkedListDeque = new LinkedList<>();
    }
}
```

### Methods of Deque:

- `add(element)` and `addLast(element)` to add element at tail and `addFirst(element)` to add element at head.
- `offer(element)` and `offerLast(element)` to add element at tail and `offerFirst(element)` to add element at head.
- `getFirst()` to see element at head and `getLast()` to see element at tail.
- `peek()` and `peekFirst()` to see element at head and `peekLast()` to see element at tail.
- `remove()` and `removeFirst()` to remove element from head and `removeLast()` to remove element from tail.
- `poll()` and `pollFirst()` to remove element from head and `pollFirst()` to remove element from tail.
- `iterator()` and `descendingIterator` for iterating the queue.
- `push(element)` and `pop()` to add and remove while using as stack.

## 2. Array Deque

- A special kind of array that grows and allows users to add or remove an element from both the sides of the queue.
- ***No capacity restrictions*** and they grow as necessary to support usage.
- ***Not thread-safe*** which means that in the absence of external synchronization, it does not support concurrent access by multiple threads.
- Null elements are prohibited in the ArrayDeque.
- ArrayDeque class is likely to be ***faster than Stack when used as a stack***.
- ArrayDeque class is likely to be ***faster than LinkedList when used as a queue***.

**Note:** We should prefer to use ArrayDeque in the places when we need either Stack or Queue or Deque.

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Ex1ArrayDequeAsStackQueueAndDeque {
    public static void main(String[] args){
        Deque<String> stack = new ArrayDeque<>();
        Deque<String> queue = new ArrayDeque<>();
        Deque<Integer> deque = new ArrayDeque<>();
    }
}
```

### Constructors in ArrayDeque:

1. **ArrayDeque():** Used to create an empty ArrayDeque and by default holds an initial capacity to hold 16 elements.
2. **ArrayDeque(collection):** Used to create an ArrayDeque containing all the elements same as that of the specified collection.
3. **ArrayDeque(numOfElements):** Used to create an empty ArrayDeque and holds the capacity to contain a specified number of elements.

### Example Usage:

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Iterator;

public class Ex2ArrayDequeCreateAndUse {
    public static void main(String[] args){
        // Create a deque using :=> ArrayDeque()
        Deque<Integer> numDeque = new ArrayDeque<>(10);

        // Add element using :=> add(element)
        numDeque.add(10);
        numDeque.add(20);
        numDeque.add(30);
        numDeque.add(40);
        numDeque.add(50);

        // Print the numbers in the Deque using :=> enhanced for loop
```

```
        System.out.print("Elements in Num Deque :");
        for (Integer element : numDeque) {
            System.out.print(" " + element);
        }
        System.out.println();

        // Clear the Deque using :=> clear()
        System.out.println("Using clear() ");
        numDeque.clear();

        // Insert at start using :=> addFirst()
        numDeque.addFirst(564);
        numDeque.addFirst(291);

        // Insert at end using :=> addLast()
        numDeque.addLast(24);
        numDeque.addLast(14);

        // Iterate in Deque using :=> iterator() and hasNext()
        System.out.print("Elements of deque using Iterator :");
        for(Iterator itr = numDeque.iterator(); itr.hasNext();) {
            System.out.print(" " + itr.next());
        }
        System.out.println();

        // Reverse Iterat using :=> descendingIterator() and hasNext()
        System.out.print("Elements of deque in reverse order :");
        for(Iterator dItr = numDeque.descendingIterator(); dItr.hasNext();) {
            System.out.print(" " + dItr.next());
        }
        System.out.println();

        // Get head element using :=> element()
        System.out.println("\nHead Element using element() : " + numDeque.element());

        // Get head element using :=> getFirst()
        System.out.println("Head Element using getFirst() : " + numDeque.getFirst());

        // Get last element using :=> getLast()
        System.out.println("Last Element using getLast(): " + numDeque.getLast());

        // Convert to Array using :=> toArray()
        Object[] arr = numDeque.toArray();
        System.out.println("\nArray Size : " + arr.length);

        System.out.print("Array elements :");
        for(int i=0; i<arr.length ; i++)
            System.out.print(" " + arr[i]);

        // Get head element using :=> peek()
        System.out.println("\nHead element using peek(): " + numDeque.peek());

        // Get head element using :=> poll()
        System.out.println("Head element poll() : " + numDeque.poll());

        // Push element using :=> push(element)
        numDeque.push(265);
        numDeque.push(984);
        numDeque.push(365);

        // Get head element using :=> remove()
        System.out.println("Head element remove : " + numDeque.remove());

        System.out.println("The final array is : " + numDeque);
    }
}
```

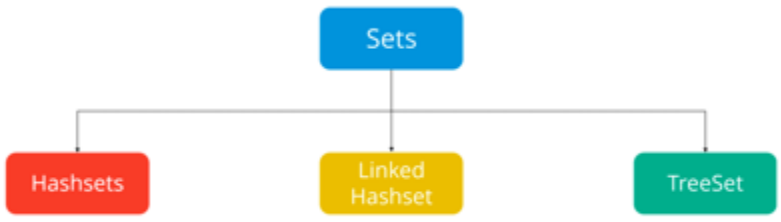
**Output:**

```
Elements in Num Deque : 10 20 30 40 50
Using clear()
Elements of deque using Iterator : 291 564 24 14
Elements of deque in reverse order : 14 24 564 291

Head Element using element() : 291
Head Element using getFirst() : 291
Last Element using getLast(): 14

Array Size : 4
Array elements : 291 564 24 14
Head element using peek(): 291
Head element poll() : 291
Head element remove : 365
The final array is : [984, 265, 564, 24, 14]
```

# Java Collections: Sets

### What are Sets ?

- A Set refers to a collection that ***can NOT contain duplicate*** elements.
- It is mainly used to model the mathematical set abstraction.
- Set has its implementation in various classes such as HashSet, TreeSetand LinkedHashSet.
- All the classes of Set interface internally backed up by Map.




### Implementations:

- In order to use the set interface, we ***need to instantiate a concrete class***.
- Following are the few implementations that can be used:
    - **Sets Interface:**
        - HashSet
        - LinkedHashSet
        - **SortedSet Interface:**
            - TreeSet
- **HashSet** doesn't maintain any kind of order of its elements.
- **LinkedHashSet** maintains the insertion order.
- **TreeSet** sorts the elements in ascending order.

```java
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

public class SetImplementations {
    public static void main(String[] args){
        Set<Integer> hashSet = new HashSet<>();
        Set<String> linkedHashSet = new LinkedHashSet<>();
        Set<String> treeSet = new TreeSet<>();
    }
}
```

# 1. HashSet

- Java HashSet class creates a collection that use a hash table for storage.
- Hashset only ***contain unique elements*** and implementsSet interface.
- Also, it uses a mechanism ***hashing*** to store the elements.
- HashSet allows **null** value.
- HashSet is an unordered collection, it does not maintain the order in which the elements are inserted.
- HashSet internally uses a **HashMap** to store its elements.
- ***NOT thread-safe***, we need to explicitly make it synchronised.

### Internal working of a HashSet

- All the classes of Set interface internally backed up by Map.
- ***HashSet uses HashMap for storing its object internally***.
- We must be wondering that to enter a value in HashMap we need a key-value pair, but in HashSet we are passing only one value.
- Actually the value we insert in HashSet acts as key to the map Object and for its value java uses a constant variable.

### Constructors in HashSet

1. **HashSet h = new HashSet();** ——> Default initial capacity is 16 and default load factor is 0.75.
2. **HashSet h = new HashSet(int initialCapacity);** —-> default loadFactor of 0.75
3. **HashSet h = new HashSet(int initialCapacity, float loadFactor**);
4. **HashSet h = new HashSet(Collection C);** ——> HashSet from another collection.

### Some Methods:

- Add element to set using `add(element)` and add all elements at once using `addAll(collection)`.
- Check if an element present in set using `contains(element)` .
- Remove elements using methods `remove(element)` , `removeAll(collection)` and **removeIf(condition)`**.
- To iterate over the loop use `iterator()`.

```java
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class HashSetCreationAndOperations {
    public static void main(String[] args){
        // Create hash set using :=> HashSet()
        Set<String> names = new HashSet<>();

        // Check if empty using :=> isEmpty()
```

```
        System.out.println("Is names set empty ? : " + names.isEmpty());

        // Add element to it using :=> add(element)
        names.add("Ravi");
        names.add("Vijay");
        names.add("Amit");
        names.add("Ajay");
        names.add("Vijay"); // duplicate will be ignored

        // Get size of the hash set using :=> size()
        System.out.println("Total names in set : " + names.size());

        // Check if an element exists using :=> contains(element)
        System.out.println("Does \"Amit\" exists in set ? : " + names.contains("Amit"));

        // Iterate loop using :=> iterator() and hasNext()
        System.out.print("Names in set are :");
        Iterator namesIterator = names.iterator();
        while (namesIterator.hasNext()){
            System.out.print(" " + namesIterator.next());
        }
        System.out.println();

        // Remove element using :=> remove(element)
        names.remove("Amit");
        System.out.println("After remove(\"Amit\") : " + names);

        // After clear()
        names.clear();
        System.out.println("After clear() : " + names);
    }
}
```

**Output:**

```
Is names set empty ? : true
Total names in set : 4
Does "Amit" exists in set ? : true
Names in set are : Vijay Ravi Amit Ajay
After remove("Amit") : [Vijay, Ravi, Ajay]
After clear() : []
```

# 2. LinkedHashSet

- Java LinkedHashSet class is a Hash table and Linked list implementation of the set interface.
- Works exactly like a **HashSet**, the only difference being it ***maintains the insertion order of elements***.
- LinkedHashSet makes use of LinkedHashMap internally for the storage.

## Constructors of LinkedHashSet

1. **LinkedHashSet():** This constructor is used to create a default HashSet.
2. **LinkedHashSet(Collection C):** Used in initializing the HashSet with the elements of the collection C
3. **LinkedHashSet(int size):** Used to initialize the size of the LinkedHashSet with the integer mentioned in the parameter.
4. **LinkedHashSet(int capacity, float fillRatio):** To initialize both the capacity and the fill ratio(load capacity).

```
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;

public class LinkedHashSetCreationAndOperations {
    public static void main(String[] args){
        Set<String> names = new LinkedHashSet<>();
        // Check if empty using :=> isEmpty()
        System.out.println("Is names set empty ? : " + names.isEmpty());

        // Add element to it using :=> add(element)
        names.add("Ravi");
        names.add("Vijay");
        names.add("Amit");
        names.add("Ajay");
        names.add("Vijay"); // duplicate will be ignored

        // Get size of the hash set using :=> size()
        System.out.println("Total names in set : " + names.size());

        // Check if an element exists using :=> contains(element)
        System.out.println("Does \"Amit\" exists in set ? : " + names.contains("Amit"));

        // Iterate loop using :=> iterator() and hasNext()
        System.out.print("Names in set are :");
        Iterator namesIterator = names.iterator();
        while (namesIterator.hasNext()){
            System.out.print(" " + namesIterator.next());
        }
        System.out.println();

        // Remove element using :=> remove(element)
        names.remove("Amit");
        System.out.println("After remove(\"Amit\") : " + names);

        // After clear()
        names.clear();
        System.out.println("After clear() : " + names);
    }
}
```

**Output:**

```
Is names set empty ? : true
```

```
Total names in set : 4
Does "Amit" exists in set ? : true
Names in set are : Ravi Vijay Amit Ajay
After remove("Amit") : [Ravi, Vijay, Ajay]
After clear() : []
```

# 3. TreeSet

- Implements the **SortedSet** interface so **NO duplicates** allowed.
- Objects in a TreeSet are stored in a ***sorted and ascending order***.
- TreeSet does not allow to insert Heterogeneous objects, will throw **classCastException** at Runtime.
- To avoid exception it must be **comparable** or should must **implement Comparable interface**, String does it but StringBuffer doesn't.
- It is basically implementation of a **self-balancing binary search tree**.
- Operations like add, remove and search take O(Log n) time and printing elements in sorted order takes O(n) time.
- Excellent for storing large amounts of sorted data which are supposed to be accessed quickly coz of its faster access and retrieval time.
- **Not thread safe** and we need to specifically synchronise it.

## Constructors of TreeSet

1. **TreeSet t = new TreeSet();** —> Create empty TreeSet which elements will get stored in default natural sorting order.
2. **TreeSet t = new TreeSet(Comparator comp);** —-> Used when external specification of sorting order of elements is needed.
3. **TreeSet t = new TreeSet(Collection col);** ——> Used when any conversion is needed from any Collection object to TreeSet object.
4. **TreeSet t = new TreeSet(SortedSet s)**; ——> Used to convert SortedSet object to TreeSet Object.

## Methods:

- Use `add(element)` to add an element and `addAll(collection)` to add collection.
- Use `first()` to get first element and use `last()` to get last element.
- Use `ceiling(given_val)` to get the smallest element greater or equal to given value.
- Use `floor(given_val)` to get greatest element smaller or equal to given value.
- Use `higher(given_val)` to get the smallest element strictly greater than given value.
- Use `lower(given_val)` to get the greatest element strictly smaller than given value.
- Use `higher(given_val)` to get the smallest element strictly greater than given value.
- Use `headSet(toElement)` to get elements of TreeSet less than the specified element.
- Use `tailSet(fromElement)` to get elements of TreeSet greater than or equal to the specified element.
- Use `subSet(fromElement, toElement)` to get elements ranging from fromElement(inclusive) to toElement(exclusive).
- Use `descendingSet()` to get reverse order view of the elements contained in this set.
- Use `iterator()` and `descendingIterator()` to loop over the element.
- Use `remove(element)`, `removeAll(collection)` and `removeIf(filterCondition)` to remove elements.
- Use `pollFirst()` and `pollLast()` to retrieve and remove lowest and smallest elements respectively.

```java
import java.util.Iterator;
import java.util.TreeSet;

public class TreeSetCreationAndOperations {
    public static void main(String[] args){
        // Create a TreeSet using :=> TreeSet()
        TreeSet<Integer> numbers = new TreeSet<>();

        // Check if empty using :=> isEmpty()
        System.out.println("Is TreeSet empty ? : " + numbers.isEmpty());

        // Add element using :=> add(element)
        numbers.add(4);
        numbers.add(1);
        numbers.add(7);
        numbers.add(9);
        numbers.add(6);
        numbers.add(5);
        numbers.add(11);
        numbers.add(17);
        numbers.add(13);

        // Calculate size using :=> size()
        System.out.println("Size of the element : " + numbers.size());

        // Smallest element greater or equal to given value using :=> ceiling(given_val)
        System.out.println("Smallest element greater or equal to 7 : " + numbers.ceiling(7));

        // Largest element smaller or equal to given value using :=> floor(given_val)
        System.out.println("Largest element smaller or equal to 12 : " + numbers.floor(12));

        // Smallest element strictly greater than than given value using :=> higher(given_val)
        System.out.println("Smallest element strictly greater than 7 : " + numbers.higher(7));

        // Largest element strictly smaller than given value using :=> lower(given_val)
        System.out.println("Largest element strictly smaller than 12 : " + numbers.lower(12));

        // All elements less than the given element using :=> headSet(toElement)
        System.out.println("All the elements less than 9 : " + numbers.headSet(9));
```

```
        // All elements greater or equal to the given element using :=> tailSet(toElement)
        System.out.println("All the elements greater or equal to 9 : " + numbers.tailSet(9));

        // All elements in range of [fromElement, toElement) using :=> subSet(fromElement, toElement)
        System.out.println("All elements in range of [6, 13) : " + numbers.subSet(6, 13));

        // Get the descending set using :=> descendingSet()
        System.out.println("Elements in reversed order : " + numbers.descendingSet());

        // Remove the elements with condition using :=> removeIf(condition)
        System.out.println("Remove all elements divisible by 3.");
        numbers.removeIf(num -> num % 3 == 0);

        // Iterate the elements using :=> iterator() or descendingIterator() and hasNext()
        Iterator<Integer> numDecIterator = numbers.descendingIterator();
        System.out.print("Remaining numbers in reverse order :");
        while (numDecIterator.hasNext()){
            System.out.print(" " + numDecIterator.next());
        }
        System.out.println();

        // Clear using :=> clear()
        numbers.clear();
        System.out.println("After clear() : " + numbers);

    }
}
```

**Output:**

```
Is TreeSet empty ? : true
Size of the element : 9
Smallest element greater or equal to 7 : 7
Largest element smaller or equal to 12 : 11
Smallest element strictly greater than 7 : 9
Largest element strictly smaller than 12 : 11
All the elements less than 9 : [1, 4, 5, 6, 7]
All the elements greater or equal to 9 : [9, 11, 13, 17]
All elements in range of [6, 13) : [6, 7, 9, 11]
Elements in reversed order : [17, 13, 11, 9, 7, 6, 5, 4, 1]
Remove all elements divisible by 3.
Remaining numbers in reverse order : 17 13 11 7 5 4 1
After clear() : []
```

# Java Collections: Maps

### What are Maps in Java ?

- The java.util.Map interface represents a mapping between a key and a value.
- The Map interface is not a subtype of the **Collection** , therefor it behaves a bit different from the rest of the collection types.

### Characteristics of Map

- A Map cannot contain duplicate keys and each key can map to at most one value.
- **HashMap** and **LinkedHashMap** allows null value but **TreeMap** doesn't.
- HashMap have NO deterministic order, LinkedHashMap maintains insertion order and TreeMap maintains the Natural Order.

### Why and When to use Maps ?

- Maps are perfect to use for ***key-value*** association mapping such as dictionaries.
- The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys.
- ***Examples:***
    - A map of error codes and their descriptions.
    - A map of zip codes and cities.
    - A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
    - A map of classes and students. Each class (key) is associated with a list of students (value).

### Implementations:

- In order to use the map interface, we ***need to instantiate a concrete class***.
- Following are the few implementations that can be used:
    - `Map Interface:`
        - HashMap
        - LinkedHashMap
        - `SortedMap Interface:`
            - TreeMap
- **HashMap** doesn't maintain any kind of order of its elements.
- **LinkedHashMap** maintains the insertion order.
- **TreeSet** sorts the elements in ascending order.

```
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
```

```java
import java.util.TreeMap;

public class MapImplementations {
    public static void main(String[] args){
        Map<String, Integer> hashMap = new HashMap<>();
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
        Map<String, Integer> treeMap = new TreeMap<>();
    }
}
```

### Map Methods:

- Put element in the map using `put(key, value)` and map(multiple elements) in map using `putAll(map)`.
- Get an element using key by `get(key)` .
- Remove an element using key by `remove(key)`.
- Check if an element present in map using `containsKey(key)` and value present using `containsValue(val)`.
- Get set view containing all the keys using `keySet()` .
- Get set view containing all the keys and values using `entrySet()`.

# 1. Hashmap

- It provides the basic implementation of the Map interface of Java.
- It can **NOT** contain duplicate keys.
- Allows **null** values and the **null** key.
- Unordered collection, does not guarantee any specific order of the elements.
- ***Not thread-safe***, we must explicitly synchronize concurrent modifications to the HashMap.

### Internal Structure:

- Internally HashMap contains an array of Node and a node is represented as a class which contains 4 fields:
    i. int hash
    ii. K key
    iii. V value
    iv. Node next
- It can be seen that node is containing a reference of its own object, so it's a linked list.

### Performance of HashMap

- Performance of HashMap depends on 2 parameters:
    i. **Initial Capacity: ** Capacity is the number of buckets whereas the *Initial Capacity* is the capacity of HashMap instance when created.
    ii. **Load Factor:** A measure that when `rehashing()` should be done
- `Rehashing` is a process of increasing the capacity, in HashMap capacity is multiplied by 2.
- `Load Factor` is also a measure that what fraction of the HashMap is allowed to fill before rehashing.
- When number of entries in HashMap increases the product of current capacity and load factor then capacity is increased *i.e.* rehashing.
- If the initial capacity is kept higher then rehashing will never be done, but by keeping it higher it increases the time complexity of iteration.
- So it should be chosen very cleverly to increase performance.
- The expected number of values should be taken into account to set initial capacity.
- Most generally preferred load factor value is 0.75 which provides a good deal between time and space costs.
- Load factor's value varies between 0 and 1.

### Constructors in HashMap

- **HashMap():** Default constructor which creates an instance of HashMap with initial capacity 16 and load factor 0.75.
- **HashMap(int initial capacity):** Creates a HashMap instance with specified initial capacity and load factor 0.75.
- **HashMap(int initial capacity, float loadFactor):** Creates a HashMap instance with specified initial capacity and specified load factor.
- **HashMap(Map map):** Creates instance of HashMap with same mappings as specified map.

### Methods:

- It has all the methods below in addition with all the provided methods by Map interface given above.
- `values()`: returns Collection view of the values in the HashMap.
- `compute(key, remappingFunction)`: Computes a mapping for given key and its current mapped value (or null if no current mapping).
- `computeIfAbsent(key, mappingFunction)`: Computes a mapping for key that is either mapped to null or doesn't exist.
- `computeIfPresent(key, remappingFunction)`: Computes a new mapping for key if value is present and not null.
- `getOrDefault(key, default)`: returns the value to which the given key is mapped, or default Value.

- `merge(key, value, remappingFunction)`: If key is either mapped to null or not exists then associates it with the given value.
- `putIfAbsent(key, value)`: Add a new key-value pair only if the key does not exist in the HashMap, or is mapped to null.
- `replace(key, value)`: Replace the key with given value.
- `replace(key, oldValue, newValue)` : Repalce the key with newValue only if oldValue matches.
- `replaceAll(function)`: Replaces all the key's value after appying fucntion on key.

## Iterating over HashMap

### 1. Iterating using :=> entrySet(), iterator() and hasNext()

```
Iterator<Map.Entry<String, Integer>> studentsIterator = students.entrySet().iterator();
while (studentsIterator.hasNext()){
    Map.Entry<String, Integer> entry = studentsIterator.next();
    System.out.print("{" + entry.getKey() + "-->" + entry.getValue() + "}, ");
}
```

### 2. Iterating using :=> enhanced for loop and entrySet()

```
for(Map.Entry<String, Integer> entry : students.entrySet()){
    System.out.print("{" + entry.getKey() + "-->" + entry.getValue() + "}, ");
}
```

### 3. Iterate using :=> forEach() and lambdaExpression

```
students.forEach((name, marks) -> System.out.print("{" + name + "-->" + marks + "}, "));
```

**Example:**

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class HashMapCreationAndOperations {
    public static void main(String[] args){
        // Create Hash Map using :=> HashMap()
        Map<String, Integer> students = new HashMap<>();

        // Check if Hash Map is empty using :=> empty()
        System.out.println("Check if students map is empty ? : " + students.isEmpty());

        // Put the key-value pairs in HashMap using :=> put(key, value)
        students.put("Badal", 78);
        students.put("Ajit", 56);
        students.put("Vishal", 95);
        students.put("Sachin", 88);
        students.put("Doval", 73);
        students.put("Manpreet", 91);

        // Get the size using :=> size()
        System.out.println("Total size of HashMap : " + students.size());

        // Check if a key exist in the using :=> containsKey(key)
        System.out.println("Is \"Vishal\" exist in students  ? : " + students.containsKey("Vishal"));

        // Check if a value present in map using :=> containsValue()
        System.out.println("Is 95 present in student marks ? : " + students.containsValue(95));

        // Get the value of a key using :=> get(key)
        System.out.println("Marks of \"Vishal\" is : " + students.get("Vishal"));

        // Get set of all the keys using :=> keySet()
        System.out.println("All the student's names : " + students.keySet());

        // Get set of all the key-value pairs using :=> entrySet()
        System.out.println("All the students and their marks : " + students.entrySet());

        // Get all the values using :=> values()
        System.out.println("All the student's marks : " + students.values());

        // Change Key and Value using :=> compute(key, remapping function)
        students.compute("Ajit", (key, val) -> val+20);
        System.out.println("Ajit Marks after recomputing value : " + students.get("Ajit"));

        // Put Key and Value if absent using :=> computeIfAbsent(key, mappingFunction)
        students.computeIfAbsent("Balram", key -> 85);

        // Get a key value using :=> getOrDefault(key, value)
        System.out.println("The marks of \"Anil \" : " + students.getOrDefault("Anil", -1));

        // Put a value if absent using :=> putIfAbsent(key, value)
        students.putIfAbsent("Anil", 68);

        // Replace the value of a key using :=> replace(key, newValue)
        students.replace("Doval", 93);

        // Iterating using :=> entrySet(), iterator() and hasNext()
        System.out.print("All the students and their marks : ");
        Iterator<Map.Entry<String, Integer>> studentsIterator = students.entrySet().iterator();
        while (studentsIterator.hasNext()){
            Map.Entry<String, Integer> entry = studentsIterator.next();
            System.out.print("{" + entry.getKey() + "-->" + entry.getValue() + "}, ");
        }
        System.out.println();

        // Remove the element using :=> remove(key)
        students.remove("Sachin");

        // Iterating using :=> enhanced for loop and entrySet()
        System.out.print("All students after \"Sachin\" removed : ");
        for(Map.Entry<String, Integer> entry : students.entrySet()){
```

```
            System.out.print("{" + entry.getKey() + "-->" + entry.getValue() + "}, ");
        }
        System.out.println();

        // Iterate using :=> forEach() and lambdaExpression
        System.out.print("All students now : ");
        students.forEach((name, marks) -> System.out.print("{" + name + "-->" + marks + "}, "));
        System.out.println();

        // Clear the map using :=> clear()
        students.clear();
        System.out.println("After clear : " + students);
    }
}
```

**Output:**

```
Check if students map is empty ? : true
Total size of HashMap : 6
Is "Vishal" exist in students  ? : true
Is 95 present in student marks ? : true
Marks of "Vishal" is : 95
All the student's names : [Manpreet, Doval, Ajit, Vishal, Badal, Sachin]
All the students and their marks : [Manpreet=91, Doval=73, Ajit=56, Vishal=95, Badal=78, Sachin=88]
All the student's marks : [91, 73, 56, 95, 78, 88]
Ajit Marks after recomputing value : 76
The marks of "Anil " : -1
All the students and their marks : {Manpreet-->91}, {Doval-->93}, {Balram-->85}, {Ajit-->76}, {Vishal-->95}, {Badal-->78}, {Sachin-->88}, {Anil-->68},
All students after "Sachin" removed : {Manpreet-->91}, {Doval-->93}, {Balram-->85}, {Ajit-->76}, {Vishal-->95}, {Badal-->78}, {Anil-->68},
All students now : {Manpreet-->91}, {Doval-->93}, {Balram-->85}, {Ajit-->76}, {Vishal-->95}, {Badal-->78}, {Anil-->68},
After clear : {}
```

# 2. LinkedHashMap

### What is a LinkedHashMap ?

- LinkedHashMap is just like **HashMap** with an additional feature of maintaining an order of elements inserted into it.
- HashMap provided the advantage of quick insertion, search and deletion but it never maintained the track and order of insertion.
- The iteration order in a LinkedHashMap is normally the order in which the elements are inserted.
- However, it also provides a special constructor using which you can change the iteration order from the least-recently accessed element to the most-recently accessed element and vice versa.
- This kind of iteration order can be useful in building LRU caches.

## Important Points

- A LinkedHashMap **NOT** contain duplicate keys.
- LinkedHashMap can have **null values** and the **null key**.
- Unlike HashMap, the iteration order of the elements in a LinkedHashMap is predictable.
- ***Not thread-safe***, we must explicitly synchronize concurrent access to a LinkedHashMap in a multi-threaded environment.

## Constructors in a LinkedHashMap

- **LinkedHashMap():** To construct a default LinkedHashMap constuctor.
- **LinkedHashMap(int capacity):** To initialize a particular LinkedHashMap with a specified capacity.
- **LinkedHashMap(map):** To initialize a particular LinkedHashMap with the elements of the specified map.
- **LinkedHashMap(capacity, fillRatio):** To initialize both the capacity and fill ratio for a LinkedHashMap.
- **LinkedHashMap(int capacity, float fillRatio, boolean Order):** To initialize both the capacity and fill ratio for a LinkedHashMap along with whether to follow the insertion order or not.
    - **True** is passed for `last access order`.
    - **False** is passed for `insertion order`.

## Methods:

- They are almost same as that of **HashMap**.

## Iterating over LinkedHashMap

- Iterating over a LinkedHashMap is similar to that of a **HashMap**.

**Example:**

```
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapCreationAndOperations {
    public static void main(String[] args){
        // Create Hash Map using :=> LinkedHashMap()
        LinkedHashMap<String, Integer> students = new LinkedHashMap<>();
```

```
        // Check if Hash Map is empty using :=> empty()
        System.out.println("Check if students map is empty ? : " + students.isEmpty());

        // Put the key-value pairs in LinkedHashMap using :=> put(key, value)
        students.put("Badal", 78);
        students.put("Ajit", 56);
        students.put("Vishal", 95);
        students.put("Sachin", 88);
        students.put("Doval", 73);
        students.put("Manpreet", 91);

        // Get the size using :=> size()
        System.out.println("Total size of LinkedHashMap : " + students.size());

        // Check if a key exist in the using :=> containsKey(key)
        System.out.println("Is \"Vishal\" exist in students  ? : " + students.containsKey("Vishal"));

        // Check if a value present in map using :=> containsValue()
        System.out.println("Is 95 present in student marks ? : " + students.containsValue(95));

        // Get the value of a key using :=> get(key)
        System.out.println("Marks of \"Vishal\" is : " + students.get("Vishal"));

        // Get set of all the keys using :=> keySet()
        System.out.println("All the student's names : " + students.keySet());

        // Get set of all the key-value pairs using :=> entrySet()
        System.out.println("All the students and their marks : " + students.entrySet());

        // Get all the values using :=> values()
        System.out.println("All the student's marks : " + students.values());

        // Change Key and Value using :=> compute(key, remapping function)
        students.compute("Ajit", (key, val) -> val+20);
        System.out.println("Ajit Marks after recomputing value : " + students.get("Ajit"));

        // Put Key and Value if absent using :=> computeIfAbsent(key, mappingFunction)
        students.computeIfAbsent("Balram", key -> 85);

        // Get a key value using :=> getOrDefault(key, value)
        System.out.println("The marks of \"Anil \" : " + students.getOrDefault("Anil", -1));

        // Put a value if absent using :=> putIfAbsent(key, value)
        students.putIfAbsent("Anil", 68);

        // Replace the value of a key using :=> replace(key, newValue)
        students.replace("Doval", 93);

        // Iterating using :=> entrySet(), iterator() and hasNext()
        System.out.print("All the students and their marks : ");
        Iterator<Map.Entry<String, Integer>> studentsIterator = students.entrySet().iterator();
        while (studentsIterator.hasNext()){
            Map.Entry<String, Integer> entry = studentsIterator.next();
            System.out.print("{" + entry.getKey() + "-->" + entry.getValue() + "}, ");
        }
        System.out.println();

        // Remove the element using :=> remove(key)
        students.remove("Sachin");

        // Iterating using :=> enhanced for loop and entrySet()
        System.out.print("All students after \"Sachin\" removed : ");
        for(Map.Entry<String, Integer> entry : students.entrySet()){
            System.out.print("{" + entry.getKey() + "-->" + entry.getValue() + "}, ");
        }
        System.out.println();

        // Iterate using :=> forEach() and lambdaExpression
        System.out.print("All students now : ");
        students.forEach((name, marks) -> System.out.print("{" + name + "-->" + marks + "}, "));
        System.out.println();

        // Clear the map using :=> clear()
        students.clear();
        System.out.println("After clear : " + students);
    }
}
```

**Output:**

```
Check if students map is empty ? : true
Total size of LinkedHashMap : 6
Is "Vishal" exist in students  ? : true
Is 95 present in student marks ? : true
Marks of "Vishal" is : 95
All the student's names : [Badal, Ajit, Vishal, Sachin, Doval, Manpreet]
All the students and their marks : [Badal=78, Ajit=56, Vishal=95, Sachin=88, Doval=73, Manpreet=91]
All the student's marks : [78, 56, 95, 88, 73, 91]
Ajit Marks after recomputing value : 76
The marks of "Anil " : -1
All the students and their marks : {Badal-->78}, {Ajit-->76}, {Vishal-->95}, {Sachin-->88}, {Doval-->93}, {Manpreet-->91}, {Balram-->85}, {Anil-->68},
All students after "Sachin" removed : {Badal-->78}, {Ajit-->76}, {Vishal-->95}, {Doval-->93}, {Manpreet-->91}, {Balram-->85}, {Anil-->68},
All students now : {Badal-->78}, {Ajit-->76}, {Vishal-->95}, {Doval-->93}, {Manpreet-->91}, {Balram-->85}, {Anil-->68},
After clear : {}
```

# 3. TreeMap

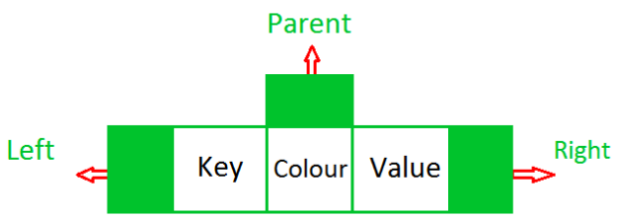
### What is a TreeMap ?

- Java TreeMap is a **Red-Black tree** based implementation of Java's Map interface.

- Entries are always sorted based on the natural ordering of the keys, or based on a custom **Comparator** provided at creation time.
- Implements the `NavigableMap` interface, which in turn extends the `SortedMap` interface.
- The **SortedMap** interface provides functionalities to maintain the ordering of keys.
- And the **NavigableMap** interface provides functionalities to **navigate** through the map.
    - ***Examples:***
        - Finding the entry just greater than or just less than the given key.
        - Finding the first and last entry in the TreeMap etc.
- Since a TreeMap implements both interface, it has the functionalities of both the `NavigableMap` as well as the `SortedMap`.

## Internal Structure:

- TreeMap is based upon tree data structure.
- Each node in the tree has:
    - 3 Variables (*K key=Key, V value=Value, boolean color=Color*)
    - 3 References (*Entry left = Left, Entry right = Right, Entry parent = Parent*)




### Important Points:

- Always sorted based on keys, sorting order follows the natural ordering if custom comprator not provided at creation time.
- Can NOT contain duplicate keys.
- Can NOT contain the **null** key, but can have **null** values.
- ***NOT thread-safe***, we must be synchronized explicitly in a multi-threaded environment.

### Constructors in TreeMap:

- **TreeMap() :** Constructs an empty tree map that will be sorted by using the natural order of its keys.
- **TreeMap(comparator) :** Constructs an empty tree-based map that will be sorted by using the comparator.
- **TreeMap(antotherMap) :** Initializes a tree map with the entries from anotherMap, which will be sorted by using the natural order of keys.
- **TreeMap(anotherSortedMap) :** Initializes a tree map with the entries from anotherSortedMap, which will be sorted in the same order as anotherSortedMap.

**Note:** TreeMap sorts the elements only on the basis of keys i.e. we can control how it will be sorted only by tweaking through keys and not through values. If we need to do it through values, we need to extract it in List and then do it.

### Methods in TreeMap:

In addition to the methods that has been given above in Map and HashMap, TreeMap also have methods as below:

- Get the firstKey using `firstKey()` and lastKey using `lastKey()`.
- Get the firstEntry using `firstEntry()` and lastEntry using `lastEntry()`.
- Get the Floor Key using `floorKey(key)` and Ceiling Key using `ceilingKey(key)`.
- Get the Floor Entry using `floorEntry(key)` and Ceiling Entry using `ceilingEntry(key)`.
- Get the Lower Key using `lowerKey(key)` and Higher Key using `higherKey(key)`.
- Get the Lower Entry using `lowerEntry(key)` and Higher Entry using `higherEntry(key)`.
- Get the Descending KeySet using `descendingKeySet()`.
- Get the Descending Map using `descendingMap()`.
- Get the headMap using `headMap(toKey)` - all the entries until the given key.
- Get the tailMap using `tailMap(fromKey)` - all the larger entries starting from fromKey.
- Get the subMap using `subMap(fromKey, toKey)` - all the entries starting from fromKey until toKey.
- Get the Descending KeySets using `descendingKeySet()` - all the keys in reverse order.
- Get the Descending EntrySets or Map using `descendingMap()` - all the entries in order.
- Retrieve and Remove the First Entry using `pollFirstEntry()` and Last Entry using `pollLastEntry()`.

***Example:***

```
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

public class TreeMapCreationAndOperations {
    public static void main(String[] args){
        // Crate using :=> TreeMap(lambdaComparator)
        TreeMap<String, Integer> students = new TreeMap<>((s1, s2) -> s1.compareTo(s2));

        // Check if TreeMap is empty using :=> empty()
        System.out.println("Check if students TreeMap is empty ? : " + students.isEmpty());
```

```
        // Put the key-value pairs in TreeMap using :=> put(key, value)
        students.put("Badal", 78);
        students.put("Ajit", 56);
        students.put("Vishal", 95);
        students.put("Sachin", 88);
        students.put("Doval", 73);
        students.put("Manpreet", 91);

        // Get the size using :=> size()
        System.out.println("Total size of TreeMap : " + students.size());

        // Check if a key exist in the using :=> containsKey(key)
        System.out.println("Is \"Vishal\" exist in students  ? : " + students.containsKey("Vishal"));

        // Check if a value present in map using :=> containsValue()
        System.out.println("Is 95 present in student marks ? : " + students.containsValue(95));

        // Get the value of a key using :=> get(key)
        System.out.println("Marks of \"Vishal\" is : " + students.get("Vishal"));

        // Get set of all the keys using :=> keySet()
        System.out.println("\nAll the student's names : " + students.keySet());

        // Get set of all the key-value pairs using :=> entrySet()
        System.out.println("All the students and their marks : " + students.entrySet());

        // Get all the values using :=> values()
        System.out.println("All the student's marks : " + students.values());

        // Change Key and Value using :=> compute(key, remapping function)
        students.compute("Ajit", (key, val) -> val+20);
        System.out.println("Ajit Marks after recomputing value : " + students.get("Ajit"));
```

```
        // Put Key and Value if absent using :=> computeIfAbsent(key, mappingFunction)
        students.computeIfAbsent("Balram", key -> 85);

        // Get a key value using :=> getOrDefault(key, value)
        System.out.println("The marks of \"Anil \" : " + students.getOrDefault("Anil", -1));

        // Put a value if absent using :=> putIfAbsent(key, value)
        students.putIfAbsent("Anil", 68);

        // Replace the value of a key using :=> replace(key, newValue)
        students.replace("Doval", 93);

        // Iterating using :=> entrySet(), iterator() and hasNext()
        System.out.print("All the students and their marks Now : ");
        Iterator<Map.Entry<String, Integer>> studentsIterator = students.entrySet().iterator();
        while (studentsIterator.hasNext()){
            Map.Entry<String, Integer> entry = studentsIterator.next();
            System.out.print("{" + entry.getKey() + "-->" + entry.getValue() + "}, ");
        }
        System.out.println();

        // Get the firstKey using :=> firstKey() and lastKey using :=> lastKey()
        System.out.println("\nFirst Key : " + students.firstKey() + ", Last Key : " + students.lastKey());

        // Get the firstEntry using :=> firstEntry() and lastEntry using :=> lastEntry()
        System.out.println("First Entry : " + students.firstEntry() + ", Last Entry : "
                + students.lastEntry());

        // Get the Floor Key using :=> floorKey(key) and Ceiling Key using :=> ceilingKey(key)
        System.out.println("Floor Key of \"Badal\" : " + students.floorKey("Badal") + ", " +
                "Ceiling Key of \"Manpreet\" : " + students.ceilingKey("Manpreet"));

        // Get the Floor Entry using :=> floorEntry(key) and Ceiling Entry using :=> ceilingEntry(key)
        System.out.println("Floor Entry of \"Badal\" : " + students.floorEntry("Badal") + ", " +
                "Ceiling Entry of \"Manpreet\" : " + students.ceilingEntry("Manpreet"));

        // Get the Lower Key using :=> lowerKey(key) and Higher Key using :=> higherKey(key)
        System.out.println("Lower Key of \"Badal\" : " + students.lowerKey("Badal") + ", " +
                "Higher Key of \"Manpreet\" : " + students.higherKey("Manpreet"));
```

```
        // Get the Lower Entry using :=> lowerEntry(key) and Higher Entry using :=> higherEntry(key)
        System.out.println("Lower Entry of \"Badal\" : " + students.lowerEntry("Badal") + ", " +
                "Higher Entry of \"Manpreet\" : " + students.higherEntry("Manpreet"));

        // Get the Descending KeySet using :=> descendingKeySet()
        System.out.println("\nKey Sets in Reverse Order : " + students.descendingKeySet());

        // Get the Descending Map using :=> descendingMap()
        System.out.println("Entry Sets in Reverse Order : " + students.descendingMap());

        // Get the headMap using :=> headMap(toKey)
        System.out.println("HeadMap until \"Manpreet\" : " + students.headMap("Manpreet"));

        // Get the tailMap using :=> tailMap(fromKey)
        System.out.println("TailMap from \"Manpreet\" : " + students.tailMap("Manpreet"));

        // Get the subMap using :=> subMap(fromKey, toKey)
        System.out.println("SubMap from \"Badal\" until \"Vishal\" : " + students.subMap("Badal", "Vishal"));

        // Get the Descending KeySets using :=> descendingKeySet()
        System.out.println("Descending KeySets : " + students.descendingKeySet());

        // Get the Descending EntrySets or Map using :=> descendingMap()
        System.out.println("Descendig EntrySets or Map : " + students.descendingMap());

        // Remove the element using :=> remove(key)
        students.remove("Sachin");

        // Iterating using :=> enhanced for loop and entrySet()
        System.out.print("\nAll students after \"Sachin\" removed : ");
        for(Map.Entry<String, Integer> entry : students.entrySet()){
```

```
            System.out.print("{" + entry.getKey() + "-->" + entry.getValue() + "}, ");
        }
        System.out.println();

        // Retrieve and Remove the First Entry using :=> pollFirstEntry() and Last Entry using pollLastEntry()
        System.out.println("Removed First Entry: " + students.pollFirstEntry() + ", Last Entry : "
                + students.pollLastEntry());

        // Iterate using :=> forEach() and lambdaExpression
        System.out.print("All students now after removing first and last entry : ");
        students.forEach((name, marks) -> System.out.print("{" + name + "-->" + marks + "}, "));
        System.out.println();

        // Clear the map using :=> clear()
        students.clear();
        System.out.println("After clear : " + students);

    }
}
```

**Output:**

```
Check if students TreeMap is empty ? : true
Total size of TreeMap : 6
Is "Vishal" exist in students  ? : true
Is 95 present in student marks ? : true
Marks of "Vishal" is : 95

All the student's names : [Ajit, Badal, Doval, Manpreet, Sachin, Vishal]
All the students and their marks : [Ajit=56, Badal=78, Doval=73, Manpreet=91, Sachin=88, Vishal=95]
All the student's marks : [56, 78, 73, 91, 88, 95]
Ajit Marks after recomputing value : 76
The marks of "Anil " : -1
All the students and their marks Now : {Ajit-->76}, {Anil-->68}, {Badal-->78}, {Balram-->85}, {Doval-->93}, {Manpreet--
>91}, {Sachin-->88}, {Vishal-->95},

First Key : Ajit, Last Key : Vishal
First Entry : Ajit=76, Last Entry : Vishal=95
Floor Key of "Badal" : Badal, Ceiling Key of "Manpreet" : Manpreet
Floor Entry of "Badal" : Badal=78, Ceiling Entry of "Manpreet" : Manpreet=91
Lower Key of "Badal" : Anil, Higher Key of "Manpreet" : Sachin
Lower Entry of "Badal" : Anil=68, Higher Entry of "Manpreet" : Sachin=88

Key Sets in Reverse Order : [Vishal, Sachin, Manpreet, Doval, Balram, Badal, Anil, Ajit]
Entry Sets in Reverse Order : {Vishal=95, Sachin=88, Manpreet=91, Doval=93, Balram=85, Badal=78, Anil=68, Ajit=76}
HeadMap until "Manpreet" : {Ajit=76, Anil=68, Badal=78, Balram=85, Doval=93}
TailMap from "Manpreet" : {Manpreet=91, Sachin=88, Vishal=95}
SubMap from "Badal" until "Vishal" : {Badal=78, Balram=85, Doval=93, Manpreet=91, Sachin=88}
Descending KeySets : [Vishal, Sachin, Manpreet, Doval, Balram, Badal, Anil, Ajit]
Descendig EntrySets or Map : {Vishal=95, Sachin=88, Manpreet=91, Doval=93, Balram=85, Badal=78, Anil=68, Ajit=76}

All students after "Sachin" removed : {Ajit-->76}, {Anil-->68}, {Badal-->78}, {Balram-->85}, {Doval-->93}, {Manpreet--
>91}, {Vishal-->95},
Removed First Entry: Ajit=76, Last Entry : Vishal=95
All students now after removing first and last entry : {Anil-->68}, {Badal-->78}, {Balram-->85}, {Doval-->93},
{Manpreet-->91},
After clear : {}
```

---

# Lambdas and Streams in Java

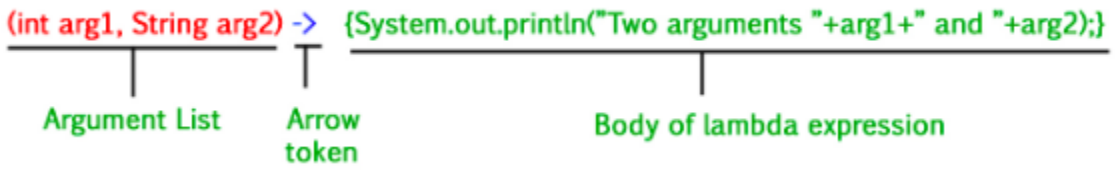
This section discussed about the the 2 major concepts in Java programming language.

- Lambdas
- Streams

## Java Lambdas

### What are lambda expressions ?

- Lambda expressions basically express instances of functional interfaces .
- An interface with single abstract method is called ***functional interface***.
- ***Example: java.lang.Runnable***.
- lambda expressions implement the only abstract function and therefore implement functional interfaces.




### Functionalities:

- Enable to treat functionality as a method argument, or code as data.
- A function that can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and executed on demand

# Java Streams

### What are Java Streams ?

- Introduced in Java 8, the Stream API is used to process collections of objects.
- A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

### Features of Java Stream :

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

## Operations on Streams

### 1. Intermediate Operations

- `map(lambdaMapper):` Used to returns a stream consisting of the results of applying the given function to the elements of this stream.

```
List<Integer> numbers = Arrays.asList(2,3,4,5);
List<Integer> squares = numbers.stream().map(e -> e*e).collect(Collectors.toList());
```

- `filter(lambdaFilterer):` - Used to select elements as per the Predicate passed as argument.

```
List<String> names = Arrays.asList("Reflection", "Collection", "Stream");
List<String> filteredNames = names.stream().filter(e -> e.startsWith("S")).collect(Collectors.toList());
```

- `sorted(lambdaComparator):` - Used to sort the stream.

```
List<String> names = Arrays.asList("Reflection", "Collection", "Stream");
List<String> sortedNames = names.stream().sorted((e1, e2) -> e2.compareTo(e1)).collect(Collectors.toList());
```

### 2. Terminal Operations :

- `collect(container):` - Used to return the result of the intermediate operations performed on the stream.

```
List<Integer> numbers = Arrays.asList(2,3,4,5,3);
Set<Integer> squareSet = numbers.stream().map(e -> e*e).collect(Collectors.toSet());
```

- `forEach(operations):` - Used to iterate through every element of the stream.

```
List<Integer> numbers = Arrays.asList(2,3,4,5);
numbers.stream().forEach(e -> System.out.print(e + " "));
```

- `reduce(binaryOperator):` - used to reduce the elements of a stream to a single value, it takes a BinaryOperator as a parameter.

```
List<Integer> numbers = Arrays.asList(2,3,4,5);
int evenSum = numbers.stream().filter(e -> e%2==0).reduce(0, (sum, i) -> sum+i);
// Here sum variable is assigned 0 as the initial value and i is added to it .
```

***Example:***

```
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class StreamCreationAndOperations {
    public static void main(String[] args){
        List<Integer> numbers = Arrays.asList(2, 3, 4, 5, 2);
        List<String> names = Arrays.asList("Reflection", "Collection", "Stream");

        // Map each element to another value using :=> map(lambdaMapper)
        List<Integer> squares = numbers.stream().map(e -> e*e).collect(Collectors.toList());
        System.out.println(squares);

        // Filter elements using :=> filter(lambdaFilterer)
        List<String> filteredNames = names.stream().filter(e -> e.startsWith("S"))
                                                   .collect(Collectors.toList());
        System.out.println(filteredNames);

        // Sort elements using :=> sorted(lambdaComparator)
        List<String> sortedNames = names.stream().sorted((e1, e2) -> e2.compareTo(e1))
```

```
                                                    .collect(Collectors.toList());
        System.out.println(sortedNames);

        // Collect from stream using :=> collect(container)
        Set<Integer> squareSet = numbers.stream().map(e -> e*e).collect(Collectors.toSet());
        System.out.println(squareSet);

        // Iterate over each element in stream using :=> forEach(operations)
        numbers.stream().forEach(e -> System.out.print(e + " "));
        System.out.println();

        // Reduce the stream to a single entity using :=> reduce(binaryOperator)
        int evenSum = numbers.stream().filter(e -> e%2==0).reduce(0, (sum, i) -> sum+i);
        System.out.println(evenSum);
    }
}
```

**Output:**

```
[4, 9, 16, 25, 4]
[Stream]
[Stream, Reflection, Collection]
[16, 4, 9, 25]
2 3 4 5 2
8
```

### Important Notes :

- A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
- Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

---

# Exceptions Handling in Java

### What is an exception in Java ?

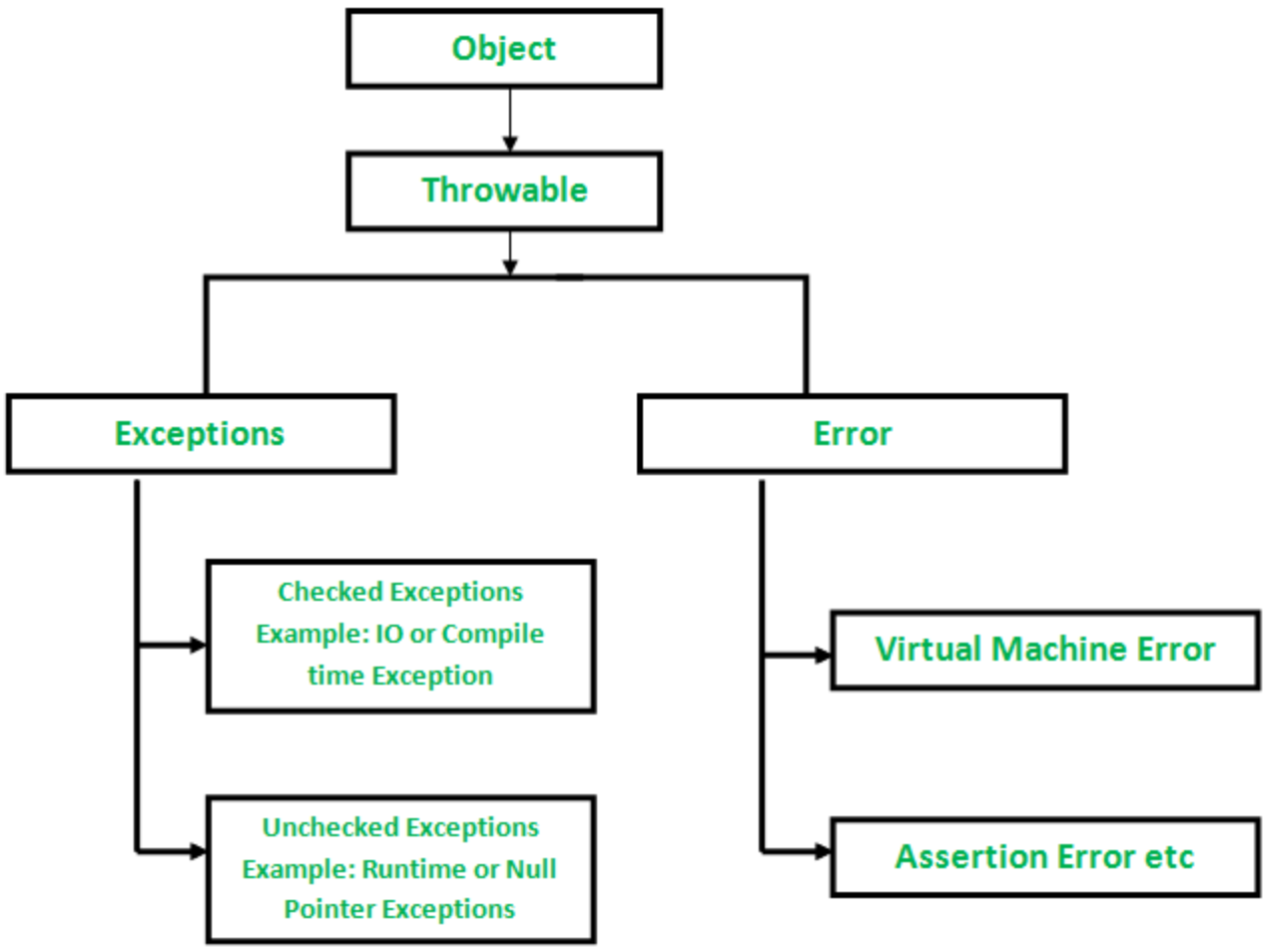
- It is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time.
- It disrupts the normal flow of the program's instructions.

### Error vs. Exception

- **Error :** indicates serious problem that a reasonable application should not try to catch.
- **Exception :** indicates conditions that a reasonable application might try to catch.

## Exception Hierarchy :

- All exception and errors types are sub classes of class `Throwable`, which is base class of hierarchy.
- **Exceptions Branch:**
  - It is used for exceptional conditions that user programs should catch.
  - *Example:* NullPointerException
- **Error Branch:**
  - Used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE).
  - *Example:* StackOverflowError




### Checked vs. Unchecked Exceptions

- `Checked Exceptions :`
  - Those exceptions are ***checked at compile time***.
  - If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using ***throws*** keyword.
  - ***Examples:*** FileNotFoundException, IOException
- `Unchecked Exceptions:`
  - Those exceptions are not checked at compiled time.
  - Exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

# Exception Handling

- In java exceptions are handled either by the programmer or by default by JVM.

## Exception Handling by JVM :

- Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM).
- The exception object contains name and description of the exception, and current state of the program where exception has occurred.
- Creating the Exception Object and handling it to the run-time system is called ***throwing an Exception***.
- There might be the list of the methods that had been called to get to the method where exception was occurred.
- This ordered list of the methods is called ***Call Stack***.
- **Now the following procedure will happen:**
  i. The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called ***Exception handler***.
  ii. The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
  iii. If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
  iv. If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to ***default exception handler*** , which is part of run-time system. This handler prints the exception information in the following format and terminates program ***abnormally***.

```
v.    Exception in thread "xxx" Name of Exception : Description
vi.   ... ..... ..  // Call Stack
```

***Example-1 :*** Handled Exception

```
class ExceptionThrown {
    // Throws the Exception(ArithmeticException), but Appropriate Exception handler is not found here
    static int divideByZero(int a, int b){
        // this statement will cause ArithmeticException(/ by zero)
        int i = a/b;
```

```
        return i;
    }

    // RunTime System searches the appropriate Exception handler in this method also but couldn't have found.
    // So looking forward on the call stack.
    static int computeDivision(int a, int b) {
        int res =0;
        try {
          res = divideByZero(a,b);
        }catch(NumberFormatException ex){ // doesn't matches with ArithmeticException
           System.out.println("NumberFormatException is occured");
        }
        return res;
    }

    // In this method found appropriate Exception handler i.e. matching catch block.
    public static void main(String args[]){
        int a = 1;
        int b = 0;

        try {
            int i = computeDivision(a,b);
        } catch(ArithmeticException ex) {  // matching ArithmeticException
            // getMessage will print description of exception(here / by zero)
            System.out.println(ex.getMessage());
        }
    }
}
```

**Output:**

```
/ by zero.
```

***Example-2 :*** Unhandled Exception

```
class ThrowsExecp {
    public static void main(String args[]){
        String str = null;
        System.out.println(str.length());
    }
}
```

**Output:**

```
Exception in thread "main" java.lang.NullPointerException
    at ThrowsExecp.main(File.java:6)
```

# Exception Handling by Programmer :

- Java exception handling is managed via five keywords: `try`, `catch`, `throw`, `throws`, and `finally`.
- **Briefly, here is how they work :**
  - Program statements that we think can raise exceptions are contained within a try block.
  - If an exception occurs within the try block, it is thrown and our code can catch this exception (using catch block) and handle it in some rational manner.
  - System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, we use the ***keyword throw***.
  - Any exception that is thrown out of a method must be specified as such by a ***throws clause***.
  - Any code that absolutely must be executed after a try block completes is put in a finally block.

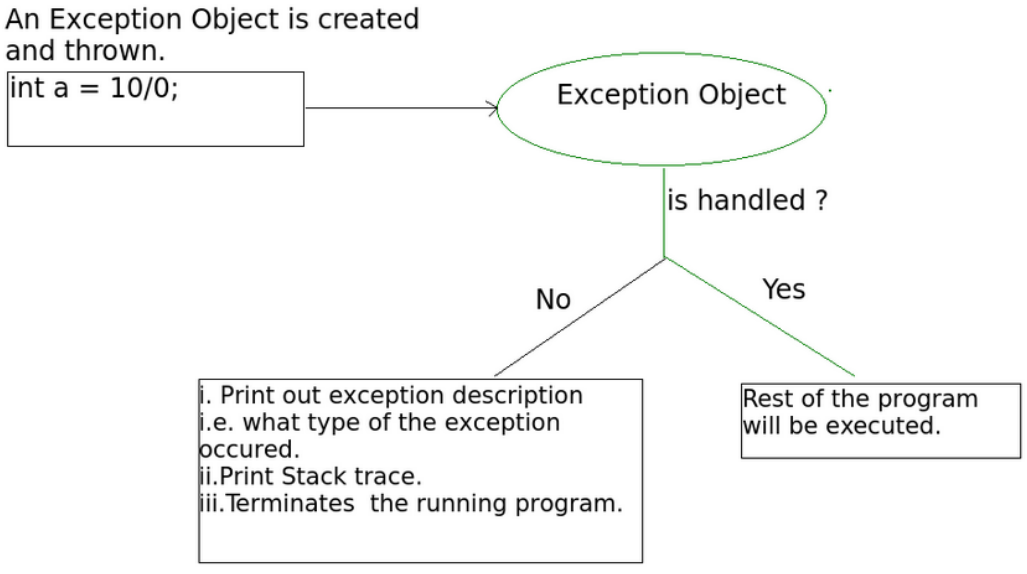
### How to handle exception ?

```
try {
                // block of code to monitor for errors the code you think can raise an exception
} catch (ExceptionType1 exOb) {
                // exception handler for ExceptionType1
} catch (ExceptionType2 exOb) {
                // exception handler for ExceptionType2
} finally {  // optional
// block of code to be executed after try block ends
}
```

Important Points :

- In a method, there can be more than one statements that might throw exception, So put all these statements within its own **try** block and provide separate exception handler within own **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put **catch** block after it. There can be more than one exception handlers. Each **catch** block is a exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of the exception that it can handle and must be the name of the class that inherits from **Throwable** class.
- For each try block there can be zero or more catch blocks, but **only one** finally block.
- The finally block is optional.It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after **try and catch blocks.** And if exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

Summary :




# Regular Expressions in Java

**What are Regular Expressions ?**

- It is an API for defining String patterns that can be used for searching, manipulating and editing a string in Java.
- Email validation and passwords are few areas of strings where Regex are widely used to define the constraints.
- Regular Expressions are provided under **java.util.regex package**.

## java.util.regex Package

This consists of 3 classes and 1 interface :

- MatchResult Interface
- Pattern Class
- Matcher Class
- PatternSyntaxException Class

# MatchResult Interface

- Used to determine the result of a match operation for a regular expression.
- It must be noted that although the match boundaries, groups and group boundaries can be seen, the modification is not allowed through a MatchResult.

Methods :

| Modifier and Type | Method | Description |
|---|---|---|
| int | `end()` | To return the offset after the last character matched. |
| int | `end(int group)` | To return the offset after the last character of the subsequence captured by the given group during this match. |
| String | `group()` | To return the input subsequence matched by the previous match. |
| String | `group(int group)` | To return the input subsequence captured by the given group during the previous match operation. |
| int | `groupCount()` | To return the number of capturing groups in this match result's pattern. |
| int | `start()` | To return the start index of the match. |
| int | `start(int group)` | To return the start index of the subsequence captured by the given group during this match. |

## 1. Pattern Class

- It is a compilation of regular expressions that can be used to define various types of patters, providing no public constructors.
- Created by invoking the `compile()` method which accepts a regular expression as the first argument and returns a pattern after execution.

### Basic Syntax:

```
Pattern pattern = Pattern.compile("geeks");
```

### Methods :

- `compile(String regex):` - compiles the given regular expression into a pattern.
- `compile(String regex, int flags):` - compiles the given regular expression into a pattern with the given flags.
- `flags():` - return the pattern's match flags.
- `matcher(CharSequence input):` - creates a matcher that will match the given input against the pattern.
- `matches(regex, input):` - compiles the given regular expression and attempts to match the given input against it.
- `pattern():` - returns the regular expression from which this pattern was compiled.
- `quote(String s):` -returns a literal pattern String for the specified String.
- `split(input):` - to split the given input sequence around matches of this pattern.
- `split(input, limit):` - to split the given input sequence around matches of this pattern.
- `toString():` - to return the string representation of the pattern.

| Modifier and Type | Method | Description |
|---|---|---|
| static Pattern | `compile(String regex)` | To compile the given regular expression into a pattern. |
| static Pattern | `compile(String regex,int flags)` | To compile the given regular expression into a pattern with given flags. |
| int | `flags()` | To return this pattern's match flags. |
| Matcher | `matcher(CharSequence input)` | To create a matcher that will match the given input against this pattern. |
| static boolean | `matches(String regex, CharSequence input)` | To compile the given regular expression and attempts to match the given input against it. |
| String | `pattern()` | To return the regular expression from which this pattern was compiled. |
| static String | `quote(String s)` | To return a literal pattern String for the specified String. |
| String[] | `split(CharSequence input)` | To split the given input sequence around matches of this pattern. |
| String[] | `split(CharSequence input,int limit)` | To split the given input sequence around matches of this pattern. |
| String | `toString()` | To return the string representation of this pattern. |

## 2. Matcher Class

- This object is used to perform match operations for an input string in java, thus interpreting the previously explained patterns.
- This too defines no public constructors and can be implemented by invoking a `matcher()` on any pattern object.

### Basic Syntax:

```
// Create a pattern to be searched
Pattern pattern = Pattern.compile("geeks");

// Search above pattern in "geeksforgeeks.org"
Matcher m = pattern.matcher("geeksforgeeks.org");
```

Methods :

| Modifier and Type | Method | Description |
|---|---|---|
| boolean | `find()` | For searching multiple occurrences of the regular expressions in the text. |
| boolean | `find(int start)` | For searching occurrences of the regular expressions in the text starting from the given index. |
| int | `start()` | For getting the start index of a match that is being found using find() method. |
| int | `end()` | For getting the end index of a match that is being found using find() method. It returns index of character next to last matching character. |
| int | `groupCount()` | It is used to find the total number of the matched subsequence. |
| String | `group()` | It is used to find the matched subsequence. |
| boolean | `matches()` | It is used to test whether the regular expression matches the pattern. |

**Note:** Pattern.matches() checks if the whole text matches with a pattern or not. Other methods are mainly used to find multiple occurrences of pattern in text.

## 3. PatternSyntaxException class

- This object of Regex is used to indicate a syntax error in a regular expression pattern and is an unchecked exception.

Methods :

| Modifier and Type | Method | Description |
|---|---|---|
| String | `getDescription()` | To retrieve the description of the error. |
| int | `getIndex()` | To retrieve the error index. |
| String | `getMessage()` | To return a multiline string containing description of syntax error and its index, the erroneous regular-expression pattern, and a visual indication of the error index within the pattern. |
| String | `getPattern()` | To retrieve the erroneous regular-expression pattern. |

Important Observations / Facts

1. We create a pattern object by calling Pattern.compile(), there is no constructor. compile() is a static method in Pattern class.
2. Like above, we create a Matcher object using matcher() on objects of Pattern class.
3. Pattern.matches() is also a static method that is used to check if given text as a whole matches pattern or not.
4. find() is used to find multiple occurrences of pattern in text.
5. We can split a text based on a delimiter pattern using split()

# Multi-threading in Java

Multi-tasking

- Multitasking is a process of executing multiple tasks simultaneously.
- We use multitasking to utilize the CPU.
- Multitasking can be achieved in two ways:
    i. **Process-based Multi-tasking (Multi-processing)**
        - Each process has an address in memory, in other words, each process allocates a separate memory area.
        - A process is heavyweight.
        - Cost of communication between the process is high.

- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

ii. **Thread-based Multi-tasking (Multi-threading)**

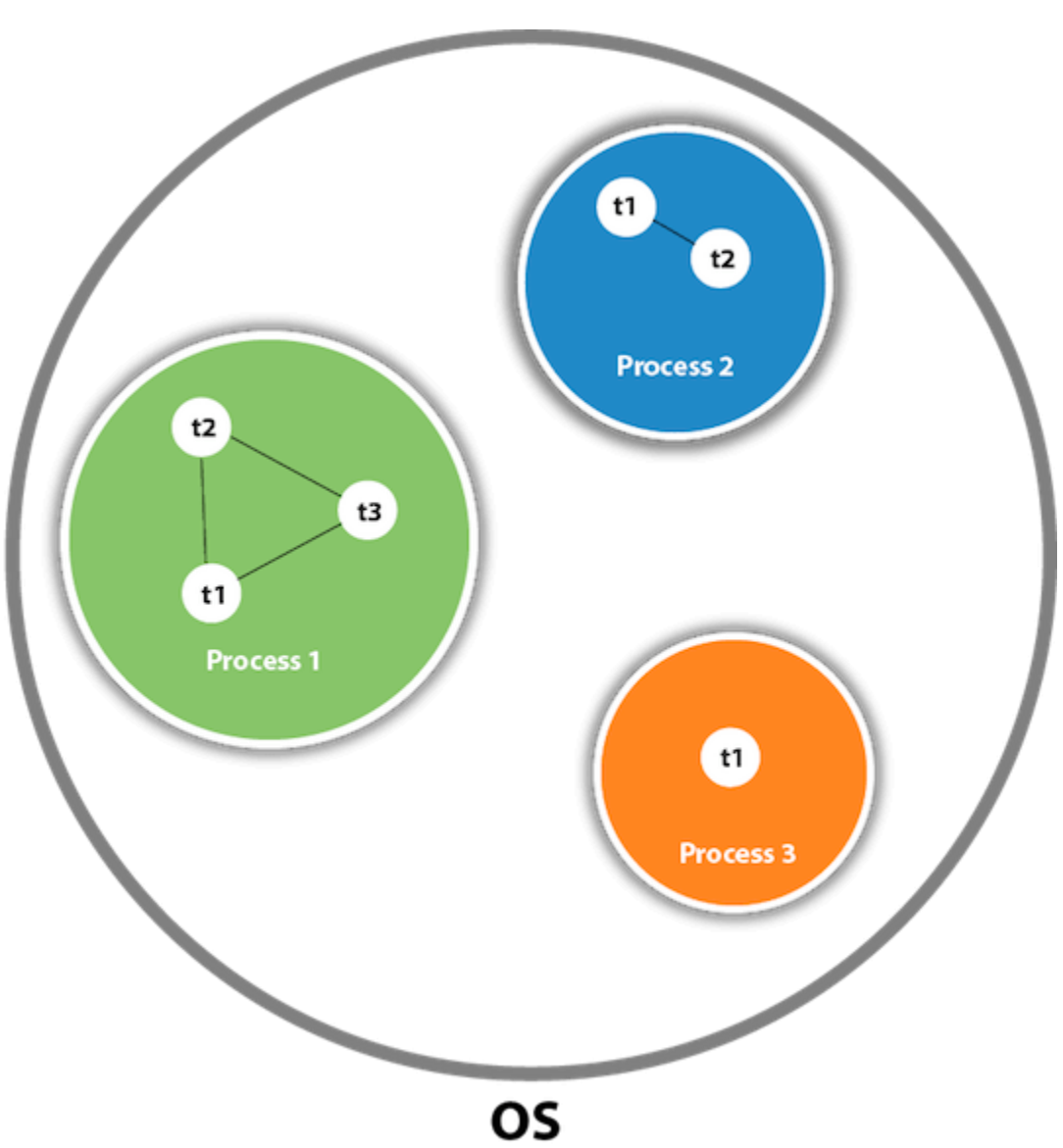
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

### What is Multi-threading ?

- Multithreading is a feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.
- Each part of such program is called a thread.
- So, threads are light-weight processes within a process.

### What are threads ?

- A thread is a lightweight subprocess, the smallest unit of processing.
- It has a separate path of execution.
- Threads are independent, if there occurs exception in one thread, it doesn't affect other threads.
- It uses a shared memory area.




[Note :** There can be multiple processes inside the OS, and one process can have multiple threads.

### How to create threads ?

Threads can be created by using two mechanisms :

1. Extending the **Thread class**
2. Implementing the **Runnable Interface**

## 1. Extending the Thread Class

- We create a class that extends the **java.lang.Thread** class.
- This class overrides the run() method available in the Thread class.
- A thread begins its life inside run() method.
- We create an object of our new class and call start() method to start the execution of a thread.
- Start() invokes the run() method on the Thread object.

```
// Java code for thread creation by extending the Thread class
class MultithreadingDemo extends Thread {
    public void run() {
```

```
        try {
            // Displaying the thread that is running
            System.out.println ("Thread " + Thread.currentThread().getId() + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args) {
        int n = 8; // Number of threads
        for (int i=0; i<n; i++) {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
}
```

**Output:**

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

## 2. Implementing the Runnable Interface

- We create a new class which implements java.lang.Runnable interface and override run() method.
- Then we instantiate a Thread object and call start() method on this object.

```
// Java code for thread creation by implementing the Runnable Interface
class MultithreadingDemo implements Runnable {
    public void run() {
        try {
            // Displaying the thread that is running
            System.out.println ("Thread " + Thread.currentThread().getId() + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args) {
        int n = 8; // Number of threads
        for (int i=0; i<n; i++) {
            Thread object = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

**Output:**

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

### Thread Class vs. Runnable Interface

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

---

# File Handling in Java

### Why File Handling is needed ?

- So far the operations using Java programs are done on a prompt/terminal which is not stored anywhere.
- But in the software industry, most of the programs are written to store the information fetched from the program.
- One such way is to store the fetched information in a **file**.

### What is File Handling in Java ?

- A **file** is a container which is used to store various types of information.
- Data is permanently stored in secondary memory by creating a file with a unique name.
- A file may consist of text, image or any other document.

### Operations performed on a file :

1. Creation of a new file
2. Opening an existing file
3. Reading from file
4. Writing to a file
5. Moving to a specific location in a file
6. Closing a file

# Packages in Java

### What are Pacakges in Java ?

- It is a mechanism to encapsulate a group of classes, sub packages and interfaces.

### Use of Packages :

- Preventing naming conflicts.
  - *Example:* there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee.
- Makes searching/locating and usage of classes, interfaces, enumerations and annotations easier.
- **Provides controlled access :**
  - protected and default have package level access control.
  - A protected member is accessible by classes in the same package and its subclasses.
  - A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as ***data encapsulation*** (or data-hiding).

### How to create a package ?

- All we need to do is put related classes into packages.
- After that, we can simply write an import class from existing packages and use it in our program.
- It is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.
- We can reuse existing classes from the packages as many time as we need it in our program.

### How packages work?

- Package names and directory structure are closely related.
- For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present *college*.
- Also, the directory *college* is accessible through **CLASSPATH** variable, i.e. path of parent directory of college is present in CLASSPATH.
- The idea is to make sure that classes are easy to locate.

### Package naming conventions :

- Packages are named in reverse order of domain names, i.e., org.geeksforgeeks.practice.
- For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

### Adding a class to a Package :

- We can add more classes to a created package by using package name at the top of the program and saving it in the package directory.
- ***We need a new java file to define a public class, otherwise we can add the new class to an existing .java file and recompile it.***

### Subpackages :

- Packages that are inside another package are the **subpackages**.
- These are not imported by default, they have to imported explicitly.
- Also, members of a subpackage have no access privileges.
- They are considered as different package for protected and default access specifiers.

```
import java.util.*;
// util is a subpackage created inside java package.
```

### Accessing classes inside a package

```
// import the Vector class from util package.
import java.util.vector;

// import all the classes from util package
import java.util.*;
```

- First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
- Second statement imports all the classes from **util** package.

```
// All the classes and interfaces of this package will be accessible but not subpackages.
import package.*;

// Only mentioned class of this package will be accessible.
import package.classname;

// Class name is generally used when two packages have the same class name.
// For example in below code both packages have date class.
// So we need to use a fully qualified name to avoid conflict
import java.util.Date;
import my.packag.Date;
```

## Types of Packages

### 1. Built-in Packages

- These packages consist of a large number of classes which are a part of Java **API**.
- Some of the commonly used built-in packages are:
  i. **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
  ii. **java.io:** Contains classed for supporting input / output operations.
  iii. **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
  iv. **java.applet:** Contains classes for creating Applets.
  v. **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , menus etc).
  vi. **java.net:** Contain classes for supporting networking operations.

### 2. User Defined Packages

- These are the packages that are defined by the user.
- First we create a directory **myPackage** (name should be same as the name of the package).
- Then create the **MyClass** inside the directory with the first statement being the **package myPackage**.

### Using Static Import

- Static import is a feature introduced in **Java** programming language that allows members ( fields and methods ) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.

```
// Note static keyword after import.
import static java.lang.System.*;

class StaticImportDemo {
   public static void main(String args[]) {
        // We don't need to use 'System.out' as imported using static.
        out.println("GeeksforGeeks");
   }
}
```

**Output:**

```
GeeksforGeeks
```