

Java

- sun microsystems => James Gosling => 1995
- java code that was able to run inside the browser was called as **Appletes**.

- Java SE **8** - 2014 (LTS)
 - Functional Programmming
 - Lamda Expressions
- Java SE **11** - 2018 (LTS)

Java SE 17 - 2021 => current version of java.

Java Platforms

1. Java Card
 - Used to develop applications on **very small devices** like smart cards.
2. Java ME (Micro Edition)
 - Is used to develop applications for **small mobile devices** which are **low in memory**.
3. Java SE(Standard Edition) (/ core java)
 - Used to develop **desktop applications**.
4. Java EE (Enterprise Edition) (advanced Java / enterprise java / web java)
 - used to develop **web applications**.

For Compilation

```
javac <name of .java file>
```

For Execution

```
java <name of .class file>
```

Main method

- in java the main method is defined as **public static void main(String args[])**

- the main method is **invoked by the JVM**.
- JVM calls this main **method directly on the classname** without creating the object that is why it is made as **static**.
- Main method does not return anything towards the JVM , that's why its return type void.
- The main method **should be accessible outside the class** that's why it is made as **public**.
- the main method takes the command line arguments and hence it has an array of String as a parameter.

main method variations

- In java we can have multiple main inside single java project.
- Every class can have a main method.
- Defining multiple main with same signature inside same class is **not allowed**.
- If we change the return type of main jvm throws an error Main method must return a value of type void in class.
- If we make the main method as non static jvm throws error Main method is not static in class.
- If we remove the public access modifier of the main then jvm throws error Main method **not found in class**.
- If we don't pass the String[] args as parameter to the main or pass any different type of parameter then jvm throws Main method not found in class.
- If we make the main method name in caps then jvm throws error Main method not found in class.
- main method **overloading is allowed**.

System.out.println()

- **System is a class** declared inside **java.lang package**
- out is a **1. static field declared inside System class**
- out is an **2. object of PrintStream class**.
- println() is a method declared **inside PrintStream class**.

through command line

1. Create directory Demo.
2. Create 2 sub directories **src** and **bin**.
3. Inside src create a "Program.java" file.

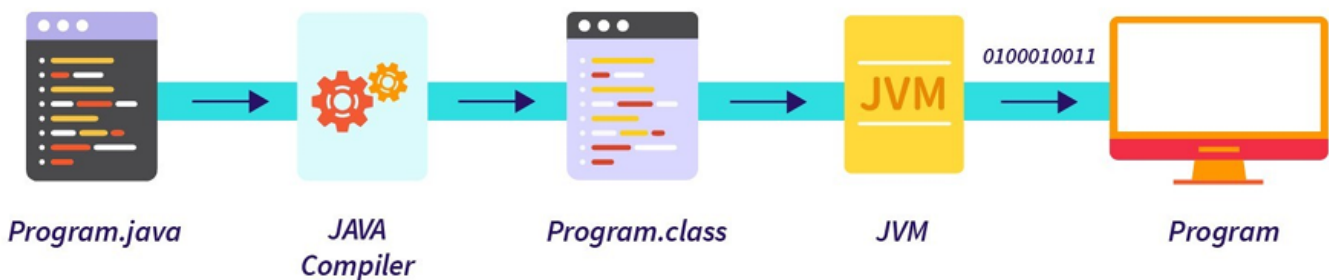
4. For compilation and execution use the below commands.

```
// from the src directory open the terminal
javac -d ../bin Program.java

// set the CLASSPATH
export CLASSPATH=../bin

// execute the code
java Program
```

Flow of Execution



JRE/JDK/JVM

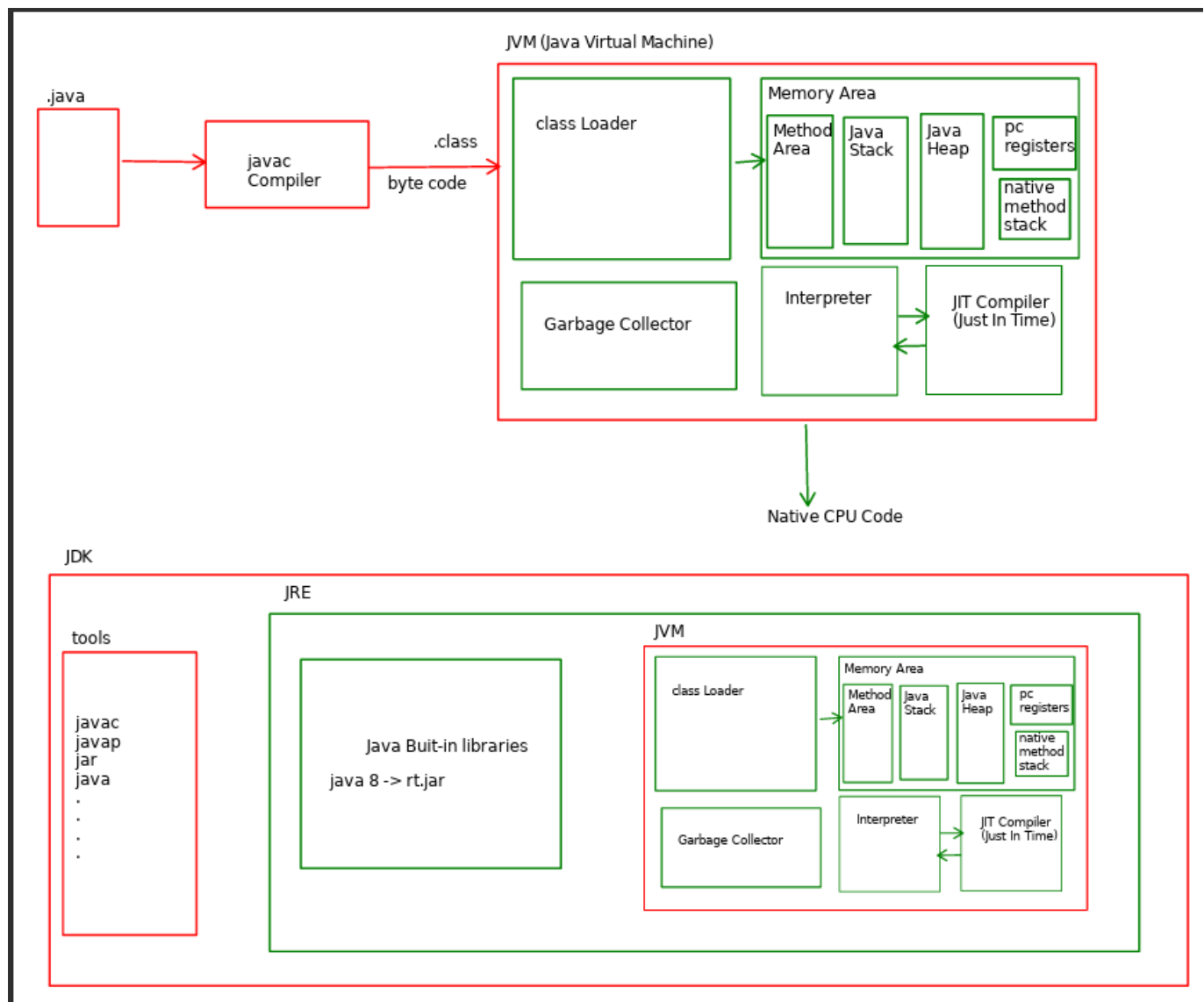
JDK (java Developemnt Kit)

- It is a software, that need to be install on developers machine.
- tools
 - javac
 - java
 - javap
 - jar
- docs
 - manuals or docs for using the tools and libraries
- libraries
 - core libraries

JRE (Java Runtime Environment)

- * rt.jar (till Java 8) [rt.jar file contains core Java API in compiled form.]
- * JVM (Java Virtual Machine)

- To deploy application, we should install JRE on client's machine.



- When we start execution of Java application then JVM starts execution of two threads:
 - Main thread** : responsible for invoking main method.
 - Garbage Collector** : responsible for deallocating memory of unused object.

Q. Why name of public class and java file name is same ?

It is the java language specification to define the public classes in its own .java file.

Q. Can we define multiple public classes in single .java file ?

No We cannot

Q. Why to make class as public ?

To maintain the visibility of the classes outside the package or in the different packages the classes need to be public.

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

public class Program01 {

    public static void main(String[] args) {
        System.out.println("Program01()");
    }

}
```

Two ways to perform input and output in java

1. using Scanner class

- It is present inside **java.util** package.
- to create the object of scanner class use below syntax.

```
Scanner sc = new Scanner(System.in);
```

2. Using Console class

- It is present inside java.io package.
- to create the object of Console class use below syntax.

```
Console console = System.console();
```

- To execute the code where Console class object is created we need to execute it through the terminal.

- execution in STS will cause `NullPointerException`

Java BuzzWords

Simple

- java have **removed** the **rarely used concept of operator overloading**.
- Java have **removed the Complexity of Pointers**.

OOP

- it supports all the major as well as minor pillars of OOP.

Compiled and Interpreted

- Java is both compiled as well as interpreted language.

Architecture Netural

- It follows **WORA - Write Once Run AnyWhere**.
- we can execute the compiled java code (.class) on any architecture.

Portable

- java is Portable because of the JVM.

Distributed

- Java Applications can be distributed on the network where multiple developers can work on the single project.
- Accesing the java objects on such distributed networks is same as that of accessing it on local machine.

Robust

- Java is robust because of its **automatic memory management**.
- It is carried out with the help of **Garbage Collector**.

MultiThreaded

- Java supports multithreading
- When we execute the java application two threads are started
 1. main Thread
 2. Garbage Collector Thread
 - * Works in the background

Secure

- you **cannot access the physical memory directly of the machine you are working on**.

- We deal directly with the **virtual memory from the JVM**.

Dynamic

- It supports **Runtime type information** which **helps java to identify objects dynamically at runtime**.

High Performance

- It is because of the **JIT Compiler**.
- When a method is called multiple times then JIT compiler compiles the code in native form and stores it into the cache.
- **So when such methods are called, jvm does not interpret them but uses the native code directly provided by the JIT compiler.**

Naming Convention

- Camel Case
 - Every first letter of the word except the first word is kept in capital.
- Pascal Case
 - Every first letter of the word is kept capital.
 - class name
 - interface
 - enums

Data types

- Data types defines 3 things
 1. **Nature**
 - What type of data can be stored inside it.
 2. **Memory**
 - How much memory is required to store the data.
 3. **Operations**
 - What all operations we can perform on that data.
- In java datatypes are divided into two categories
 1. **Primitive types** (Value Types)
 - boolean (1 bit)
 - char (2 bytes)
 - byte (1 byte)
 - short (2 bytes)
 - int (4 bytes)
 - long (8 bytes)

float (4 bytes)
double (8 bytes)

Data Types		
Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits. Range is 1.40239846e-45f to 3.40282347e+38f
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits. Range is 4.94065645841246544e-324 to 1.79769313486231570e+308
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

2. **Non Primitive types** (Reference Types)

class
interface
enum
Array

Variables


- It is a **container that can store a value***.
- The variable can be cretaed of primitive as well as non primitive type.
- The variable created of **non-primitive type is called as reference**.
- The variable created can be assigned with another variables or with the constant values.

Literals

- the **constant values used to initialize the variables** are called as literals.
- java have defined below six literals
 1. Integral Literals
 2. Floating Point Literals
 3. String Literals
 4. Character Literals
 5. Boolean Literals
 6. null Literal


```
int num1 = 10; // Intergral Literal
float salary = 1000000.123f; // Floating point Literal
String name = "sunbeam"; // String Literal
char ch = 'a'; // character Literal
boolean status = true; // Boolean Literal
Scanner sc = null; //null Literal
```

Operators

- Java have classified the operators into below categories
 1. Arithmetic Operators => +, -, *, /
 2. Assignment Opera
 3. tors => =, +=, -=, etc..
 4. Comparaision Operators => ==, <, >, <=, >=, etc..
 5. Logical Operators => &&, ||, !
 6. Bitwise Operators => &, |, ~, etc
 7. Misc Operators => Ternary Operator (? , dot Operator (.)

Narrowing/Widening

- Keeping the narrower type of data into the wider type is called as Widening.
- Keeping the wider type of data into narrower type is called as Narrowing.
- At the time of **narrowing, explicit typecasting is mandatory**.
- Narrowing may cause **data loss**.

```
int num1 = 10;
double num2 = num1; // Widening

double num3 = 123.45;
int num4 = (int)num3; // Narrowing
```

Wrapper classes

- All the primitive types are not classes but java have given classes for all such primitive types.
- These classes are called as Wrapper classes

- Use of Wrapper classes
1. **For conversion** from primitive type to respective reference type.
 2. To get the **SIZE , Range(Max and min) value** of a primitive datatype.
 3. to use **helper methods** provided by these classes.
 4. **Java collection** cannot store data of primitive types it can only store reference types.

Boxing/Unboxing

- **Converting value type into reference type** is called as boxing.
- If the boxing is done automatically **without any helper methods** then it is called auto-boxing.

```
int num1 = 20;
Integer i1 = new Integer(num1); // Boxing
Integer i2 = Integer.valueOf(num1); // Boxing
Integer i3 = num1; // Auto boxing
```

- Converting the reference type into value type is called as unboxing.
- If the unboxing is done automatically without any helper methods then it is called as auto unboxing.

```
Integer i1 = new Integer(10);
int num1 = i1.intValue();// Unboxing
int num2 = i1; // auto unboxing
```

Command Line Arguments

```
java Program 10 20
```

Control Structures

- In java all the statements are executed one after the other
- we can control the flow of statements using control statements
- Types of control statements

1. Decision Making Statements

- if statement
- switch

2. Loop Statements

- do..while
- while
- for
- for-each

3. Jump Statements

- break
- continue

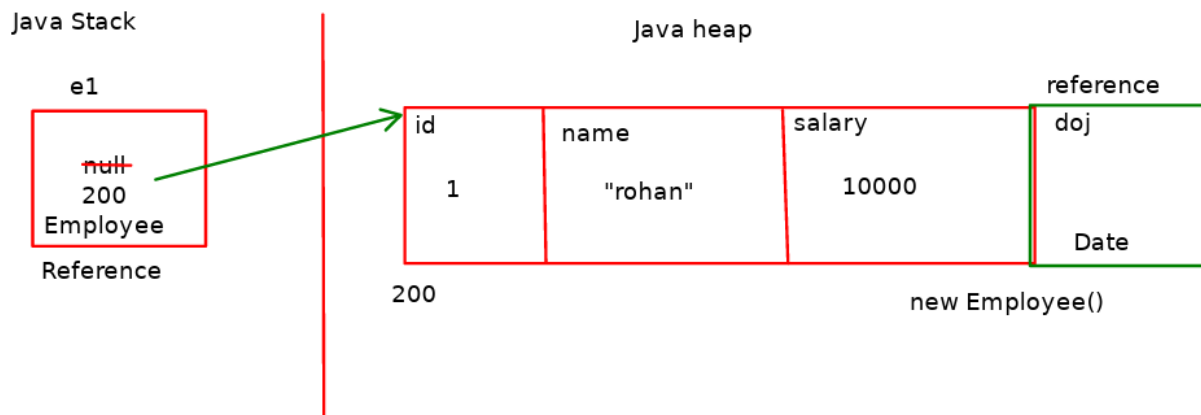
Java Method

- In java we ***cannot define the functions globally.***
- If we want to define a function that must be defined inside the class only.
- the functions that are defined inside the class are called as methods in java.
- Methods can be defined as staic or non static.
- methods can return something or it can return void.
- methods can have parameters or it can be parameterless.

Reference

- ***Variable created of a class is called as reference in java.***
- ***reference points to the object of the class.***
- local references gets created on java stack.

- references declared as fields inside the class gets the memory on the heap section.



Class

Object

forEach loop

```
for(int element:arr)
    System.out.println(element);
```

Scanner

- A class (`java.util.Scanner`) that represents text based parser.
- It can parse text data from any source.
- Scanner is a **final class** declared in **`java.util`** package.
- Methods of Scanner class:
 1. `public String nextLine()`
 2. `public int nextInt()`
 3. `public float nextFloat()`
 4. `public double nextDouble()`.

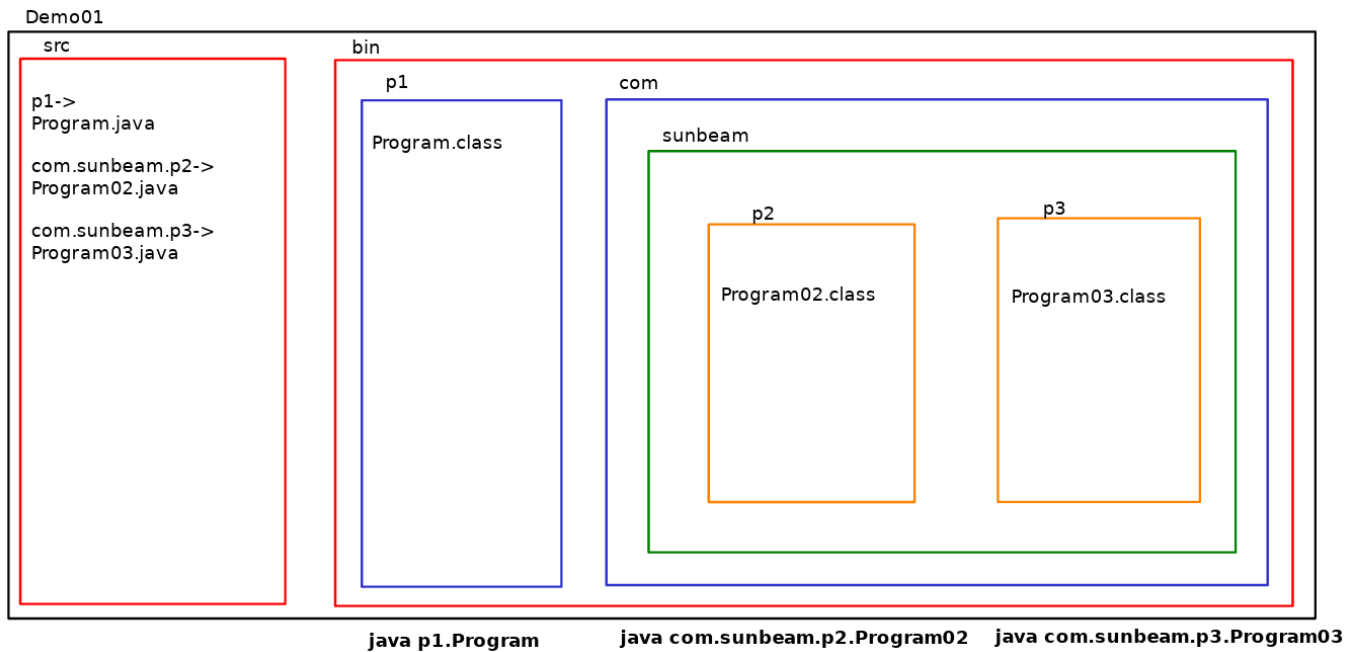
Package

- It is a **container that is used to group functionally equivalent or related types together**.
- It is also **used to avoid name ambiguity**.
- The types defined inside the package are called as **packaged types**.
- the types that are not defined inside the package are called as **unpackaged types**.
- If we define any type without package then it is considered as member of **unnamed/default package**.
- For every package that we are going to mention for the types a physical directory will be created for it.
- package names are based on company name,module name or the domain of the company.
- the domain of the company is mostly used for package names in reverse order.
- eg -> sunbeaminfo.com package name -> com.sunbeaminfo.module
- eg-> java packages java.lang java.util java.io java.sql
- The type that you want to keep inside a package can be specified by writing the package declaration inside the .java file.
- the **package declaration** should be the **first statement inside the .java file**.

```
package com.sunbeam.p2;
```

- packages are used to **organise the code**.
- package names should be kept in smallcase.
- To access the types from different packages we have to **import** those types from that package.
- to import them add the import statement with Fully Qualified classname of the type that you want to import.

```
import p2.Time;
```



AccessModifiers

1. private

- The members of the class will be accessible within the class only.
- They are ***not accessible outside the class.***

2. default (package level private)

- They are accessible within the class directly.
- They are ***accessible within the package in other class on its class object.***
- They are ***not accessible in other packages.***

3. protected

- They are accessible within the class directly.
- They are ***accessible within the package in other class on its class object.***
- They are ***accessible in other packages only in their subclasses directly.***
- They are accessible inside any of their subclasses directly.

4. public

- They are accessible everywhere in all packages and class on its class object.
- They are directly accessible inside the subclasses.

- We can also make classes as public or default.
- default classes are accessible only within the same package.
- these classes are not accessible outside the package.
- If classes are made as public they are accessible within the package and also outside the package.

Other Modifiers :

- abstract
- final
- interface
- native
- static
- strict
- synchronized
- transient
- volatile

this reference

- It is a reference that is internally passed to all the non static methods of the class.
- this reference is **constant reference** i.e once it is initialized by an object you cannot change it to point at other object inside the methods.
- using this reference is optional.
- it can be **used to identify the difference between the local variables and the class fields.**
- This is also used to point at the respective methods of the same class.

Types of methods (Demo06)

There are 4 types of methods

1. Constructor

- It is **used to initialize the state of an object.**
- {not for creating object}.
- 2. Setters
 - To set/change value of individual field of the class.
- 3. Getters
 - to get/read value of individual field of the class.
- 4. Facilitators
 - to provide business logic / operations.

Constructor Chaining

- If we want **to call one constructor from another constructor of the same class** then we can perform constructor chaining.
- to perform ctor chaining we have to use **this** statement.
- **this statement must be the first statement inside the constructor.**
- we can use it to call parameterized ctor from our parameterless ctor.

```
public class Time {  
    private int hours;  
    private int minutes;  
  
    public Time() {  
        this(10, 10); // Constructor Chaining  
    }  
  
    public Time(int hours, int minutes) {  
        this.hours = hours;  
        this.minutes = minutes;  
    }  
}
```

Object Initializer

- In java, we can initialize the objects in 3 ways.
 1. Field Initializers
 2. Object Initializers
 - It is a block that we can write inside a class where we can initialize the fields of the class.
 - this block/object initializers gets called for every object that we create.
 - if multiple object initializers are defined then they called in the same way as they are defined.

3. Constructor.

```
class Date{
int day = 1; // Field Initializer
    int month;
    int year;

    // Object initializers
    {
        this.month = 2;
    }

    // Constructor
    Date(){
        this.year = 2000;
    }
}
```

Final

In java we can make

1. variable
2. field
3. method
4. class

final variable

- final local variables can be initialized or it can be assigned with the value later.
- **once initialized or assigned we cannot change the value inside it.**

```
public static void main(String[] args) {
    //final int num = 10; // OK
    final int num; // OK

    System.out.println("Hello");

    num = 10; // OK
    //num = 20; // NOT OK
}
```

```
//final Date d1 = new Date(); // OK
final Date d1;
d1 = new Date(); // OK
//d1 = new Date(); // NOT OK
}
```

final field

- Final fields can be initialized inside
 - field initializer
 - object initializer
 - constructoronce they are initialized we cannot change the value inside it.

```
public class Date {
    final private int day = 1; // Field Initializer
    final private int month;
    final private int year;

    //object initializer
    {
        month = 1;
    }

    //Constructor
    Date(){
        year = 2001;
    }

    public void displayDate() {
        // day = 2; // NOT OK
        // month = 2; // NOT OK
        // year = 2002; // NOT OK

        System.out.println("Date - "+day+"/"+month+"/"+year);
    }
}
```

final method

final class

Array

- Advantage Of Array

1. We can access elements of array randomly.

- Disadvantage Of Array

1. We can not resize array at runtime.
2. It requires continuous memory.
3. Insertion and removal of element from array is a time consuming job
4. Using assignment operator, we can not copy array into another array.
5. Compiler do not check array bounds(min and max index).

- Arrays are of 3 types in java

1. Single Dimension Array

2. MultiDimension Array

- Array with more than 1 dimension is called as multidimensional array.
- For multidimensional array the size of both the dimensions are fixed.

3. Ragged Array

- Array with more than 1 dimension is called as multidimensional array.
- For multidimensional array if the size of second dimension is not fixed or vary then such an array is called as ragged array.

```
// Single Dimension Array
Point arr1[] = new Point[5];

//MultiDimensional Array
Point arr2[][] = new Point[2][3];

// Ragged Array
Point arr3[][] = new Point[2][];
arr3[0] = new Point[2];
arr3[1] = new Point[3];

public static void main(String[] args) {
    int arr1[][] = new int[2][]; // Ragged Array
    arr1[0] = new int[2];
    arr1[1] = new int[3];

    arr1[0][0] = 10;
    arr1[0][1] = 20;
    arr1[1][0] = 30;
    arr1[1][1] = 40;
    arr1[1][2] = 50;

    for(int row = 0; row < 2; row++) {
```

```
        for(int col = 0; col<arr1[row].length ; col++)
            System.out.println("element = "+arr1[row][col]);
    }

    System.out.println();
    System.out.println("using for-each");

    for(int arr[:arr1)
        for(int element:arr)
            System.out.println("element = "+element);

}
```

Variable Arity/Argument Method

```
public static void add(int ...args) {
    int result = 0;
    for(int element: args)
        result= result + element;

    System.out.println("Addition = "+ result);
}

add(10,20,30,40);
int arr[] = {10,20,30,40,50};
add(arr);
```

- If we want to pass multiple/variable no of arguments of same type to the method then we can define an variable arity method.
- The parameter args is **internally working as an array**.
- the arguments passed to this method at the time of method call will be converted in an array and will be passed to it.
- we can use a for-loop or for-each loop to iterate over this array.

Method Parameters

- We can pass arguments to the method that we call.
- For primitive types the arguments are always passed by value
- For non primitive types the arguments are always passed by reference
- In the pass by reference, the references gets copied.

- when the references are copied then both the references point at the same object.
- if changes in the object is done by using any reference then it changes the original object values.
- so using any reference that point at that object, if we try to read/display the state, we will always get the updated state of the object.

Static

- static means **shared variables**.
- The static fields are designed to **shared across multiple objects**.
- **In java we cannot make local variables as static.**
- In java , we can make
 1. Field as static
 2. method as static
 3. block as static
 4. import static

Static Field

- We can declare fields inside the class as static.
- Static fields get the memory on the method area.
- memory is allocated at the time of class loading.
- static fields are designed to be shared across multiple objects.
- Static fields can be initialized using
 1. static field initializer
 2. static block
- static fields are designed to be accessed on classname using . operator.
- The static fields can also be accessed on object of the class using . operator.
- It is however discouraged to use object of the class to access as it creates the confusion between static and non fields of the class.

```
//private static double roi = 5; // static field initializer
private static double roi;

// static block
static {
    roi = 6;
}
```

Static Block

- These blocks are used to initialize the static fields of the class.
- It is a block similar to the object initializer block but with static keyword.
- It gets executed only once at the time of class loading.
- If multiple static blocks are present then they will be executed in the same sequence in which they are declared.

Static Method

- These methods are designed to be accessed on classname using . operator.
- The static methods can also be accessed on object of the class using . operator.
- However it is discouraged to access them on class object as it creates confusion between static and non static methods.
- static methods cannot access non static fields of the class, as it does not get this reference.
- static methods can access only static fields of the class.
- static methods are generally used to design factory methods.

```
public static void changeRoi() {  
    System.out.print("Enter the new roi - ");  
    roi = new Scanner(System.in).nextDouble();  
}
```

Static import

- In java we can access all the static methods directly inside another static methods of the same class.
- If we want to access the static methods of different class directly without using the classname and . operator then use static import.

```
//import static com.sunbeam.BankAccount.changeRoi;  
//OR  
import static com.sunbeam.BankAccount.*;
```

In java, & is not allowed, hence we ***cannot pass by reference the "primitive types"*** ,

NP types are by default always passed as reference in java.

To pass P type as reference in java, either wrap them in wrapper class, or in collection or make them as fields of a class.

Singleton Design Pattern

- Singleton design pattern is a **way in which we create only one instance of our class**.
- we cannot create multiple instances.
- If we try to create second one, we get reference of the first object only.
- Steps for creating singleton class ->
 1. To make a class as singleton the first step to **make all the provided constructors as private**.
 2. Provide a **static field** of the **same type as that of class**.
 3. **initialize this field with the class object inside static block**.
 4. **provide a getter method** which will **return this singleton object**.

```
package com.sunbeam;

public class Singleton {
    //step-2 -> declare a private static field of same type as that of the class
    private static Singleton singleton;

    //step-3 -> write static block to initialize the static reference
    static {
        singleton = new Singleton();
    }

    // Step-1 -> Make the ctor as private
    // We cannot create the object of this class outside this class
    private Singleton() {
        System.out.println("Inside Ctor");
    }

    // step-4 -> provide a getter method that will return the singleton reference.
    public static Singleton getSingleton() {
        return singleton;
    }
}
```

```

    }

    // public static Singleton getInstance() {
    //     if(ref==null)
    //         ref = new Singleton();
    //     return ref;
    // }
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    public static void main(String[] args) {

        Singleton s1 = Singleton.getSingleton();
        Singleton s2 = Singleton.getSingleton();
    }

```

Hirerachy

Association

Inheritance

- We perform inheritance in java using **extends** keyword.
- **a class can extend only one class.**
- a class cannot extends multiple classes.
- Multiple implementation inheritance (class inheritance) is not supported in java.
- However **multiple interface inheritance is allowed** in java.

```

public class Employee extends Person // inheritance
{

}

```

super keyword

- Super is a keyword in java.
- It is **used to invoke the super class methods.**
- It is used to invoke the constructor of super class from sub class constructor.
- **super statement must be the first statement inside the constructor.**

- super is also used to invoke the super class methods when the methods of superclass are hidden inside the subclass.
- If the superclass and the subclass method is same(method overriding is done) then to invoke the hidden method of superclass inside subclass we should use this super keyword.

```
@Override
public void acceptData() {
    super.acceptData();
}
```

Method Overriding

Method overriding is done when

1. super class method is **100% incomplete**.
2. super class method is **partial complete**.
3. **if we require the implementation of the superclass method inside subclass with complete different requirement.**

Rules of method overriding

1. the name of super class and subclass method should be same.
2. no of parameters and type of parameters should also be same (signature should be same).
3. **The return type of the subclass method can be the subclass type of the return type of superclass method.**
4. **The exception list in subclass method should be same or it can be subset of exception list mentioned for the superclass method.**
5. **We can change the access modifier for the overridden method in subclass, however it should be of wider visibility type than that of superclass method.**

Types of Inheritance

```
// Single Inheritance
class A{
}
class B extends A{
}
```

```
// Multiple Inheritance
class A{
}
class B{
}
//class C extends A,B {}// NOT Allowed in Java
interface A
{
}
interface B
{
}
interface C
{
}
class D implements A,B,C // Allowed
{
}
```

```
// Multilevel Inheritance
class A{
}
class B extends A{
}
class C extends B{
}
```

```
// Hirerachical Inheritance
class A{
}
class B extends A{
}
class C extends A{
}
```

- If you mix any two interitances it is hybrid inhetitance.

Upcasting

- Keeping the **object of subclass inside reference of superclass** is called as upcasting.
- Using the superclass reference we can call the overridden methods of sub class.
- using the superclass reference we cannot call the individual methods that belong to sub class.

- It is because of **object slicing**.

Downcasting

- Converting the **reference of superclass into the subclass reference** is called as downcasting.
- At the time of downcasting **explicit typecasting is mandatory**.
- To call the individual methods given by the subclass we need to convert the reference of superclass into subclass.
- If upcasting is not done, i.e if the superclass reference is pointing at a different object then your downcasting fails.
- In java when downcasting fails we get an exception called as **ClassCastException**.

```
Employee employee = new Manager(); // upcasting

employee.acceptData();
employee.displayData();
//employee.calculateTotalSalary(); // NOT OK -> object slicing

Manager manager = (Manager)employee; // Downcasting
manager.calculateTotalSalary();
```

Dynamic Method Dispatch

- **When such super class ref is used to invoke the overriding method, then the decision to send the method for execution is taken by JRE & not by compiler.**
- In such case overriding form of the method(sub-class version) will be dispatched for execution.
- This is called as Dynamic Method Dispatch.
- **Javac(compiler) resolves the method binding by the type of the reference & JVM resolves the method binding by type of the object it's referring to.**

instanceof

- It is an **operator used to check for the instance of subclass object inside superclass reference**.
- It **returns true if the instance is present** and **false if instance is not present**.
- To avoid ClassCastException it is recommended to check for the instance using instanceof operator.

```
Employee employee = new Manager();
if(employee instanceof Manager)
{
    Manager manager = (Manager)employee;
    manager.calculateTotalSalary();
}
```

- Downcasting is not always safe, as we explicitly write the class names before doing downcasting, **It won't give an error at compile time but it may throw ClassCastException at runtime**, if the parent class reference is not pointing to the appropriate child class.
- To remove ClassCastException we can use instanceof operator to check right type of class reference in case of downcasting.

final

- **After storing value, if we don't want to modify it then we should declare variable final.**
- We can provide value to the final variable either at compile time or run time.
- In Java, we can declare **reference final**, But we can not declare instance final.

1. variable

- **once initialized cannot change the value.**

2. field

- **once initialized cannot change the value.**

3. method

- If implementation of method in superclass is **100% complete** then make such method as final.
- final methods **cannot be overridden inside the subclass.**
- we can make all the getters and setters of the superclass as final.

4. class

- If the implementation of superclass is **100% complete** then make the class as final.
- we **cannot extend final classes.**
- eg
 - java.lang.System
 - java.lang.Integer

```
public final void method1() {

}

////////////////////////////////////
final class Base {
}
```

Object class

- Object class is the **superclass of all the classes in java**.
- It directly or indirectly inherited in all the java given classes or in the classes that we are going to define.
- 11 Methods of object class
 - public **final** Class<?> **getClass()**
 - public int **hashCode()**
 - public **boolean equals(Object obj)**
 - protected **Object clone() throws CloneNotSupportedException**
 - public String **toString()**
 - public **final** void **notify()**
 - public **final** void **notifyAll()**
 - public **final** void **wait(long timeout)** throws InterruptedException
 - public **final** void **wait(long timeout,int nanos)** throws InterruptedException
 - public **final** void **wait()** throws InterruptedException
 - protected void **finalize()** throws Throwable
- All the methods of object classs are inherited into the subclasses.
- final methods declared inside the object class cannot be overridden.
- Object class have only **1 constructor which is a parameterless ctor.**

toString()

- It is a non final method declared inside object class.
- **This method is provided by the object class so that we can override it to return the state of an object in string format in human readable form.**
- It is recommended to override toString() in all the subclasses.

- the object class `toString()` returns the state of an object in the form of **fully qualified class name @hashcode**.

```
public String toString()
{
    return getClass().getName() + '@' + Integer.toHexString(hashCode());
}
```

- To print the state of object in human readable form return the state in string format from the `toString()` by overriding it into the subclass.

```
@Override
public String toString()
{
    return day+"/"+month+"/"+year;
}
```

equals()

- It is **reflexive**: for any non-null reference value `x`, `x.equals(x)` should return true.
- It is **symmetric**: for any non-null reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
- It is **transitive**: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
- It is **consistent**: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return false.

```
@Override
public boolean equals(Object obj) {
    if (this == obj) // d1.equals(d1)
        return true;
    if (obj == null) //d1.equals(null)
        return false;
    if (!(obj instanceof Date)) //d1.equals(e)
        return false;
    Date other = (Date) obj; //d1.equals(d2)
```

```
        return day == other.day && month == other.month && year == other.year;
    }
```

hashCode()

- It is a method of the object class which is **responsible to generate hashCode for every object**.
- If the **state of two objects is equal then their hash value should also be same**.
- It is recommended to override the hashCode() at the time of **overriding of equals** method.

```
@Override
public int hashCode() {
    return Objects.hash(day, month, year);
}
```

Abstract

- In Java, abstract keyword is used for
 - abstract method
 - abstract class.
- It is an accesss modifier.
- We can make the method and class as abstract.
- If the implementation of the method is **100% incomplete**, i.e we dont kow the implementation of the method in superclass.
- Such methods can only be declared without the body with the help of abstract keyword.
- such methods are called as abstract methods.
- **If class contains one or more abstract methods, then class must be declared as abstract. Otherwise compiler raise an error..**
- **we cannot create object of the abstract, we can only create the reference.**
- If we want to create the object of abstact class, inherit such class inside a subclass, and we can then create the object of the subclass and assign it ti the reference of the super class.

- **To create the object of the subclass, it is compulsory for the subclass to override all the abstract methods of the superclass.**
- Abstract classes are used to keep the **method design same across related types**.
- An abstract class can have **zero or more** abstract methods.
- Abstract "**classes**" can have "**fields**" as well as "**constructors**".
- The **abstract "method" cannot be "private", "final", or "static"**.
- Abstract classes can have abstract as well as non abstract methods.
- The super-class abstract methods must be overridden in sub-class; otherwise sub-class should also be marked abstract.
- The abstract methods are forced to be implemented in sub-class. It ensures that sub-class will have corresponding functionality.
- Example:
 - java.lang.Number
 - java.lang.Enum
- Example: abstract methods declared in **java.lang.Number class** are:
 - abstract int intValue();
 - abstract float floatValue();
 - abstract double doubleValue();
 - abstract long longValue();

```
abstract public class Employee {
    int empid;
    String name;
    double salary;

    public void acceptData(Scanner sc) {
        System.out.print("Enter id - ");
        empid = sc.nextInt();
        System.out.print("Enter name - ");
        name = sc.next();
        System.out.print("Enter Salary - ");
        salary = sc.nextDouble();
    }

    public abstract void calculateTotalSalary();

    @Override
    public String toString() {
        return "Employee [empid=" + empid + ", name=" + name + ", salary=" +
salary + "]\n";
    }
}
```



```
}
```

Fragile base class problem

- *If changes are done in "abstract super-class", "non-abstract methods" then other methods(from same class or base class) are using them through this or super keywords, then it is necessary to modify and recompile all its sub-classes. This is called as "Fragile base class problem".*
- This can be overcome by using **interfaces**.

Interface (Java 7 or Earlier)

- **Interfaces are used to define standards/specifications.** A standard/specification is set of rules.
- **Interfaces are "immutable"** i.e. once published interface should not be modified.
- Interfaces contains **only method declarations**. All **methods** in an interface are **by default "abstract" and "public"**.
- They define a "contract" that is must be followed/implemented by each sub-class.
- Interfaces enables **loose coupling between the classes** i.e. as long as you use the interface you are unaware of the implementation you choose, you can choose any implementation.{i.e. into interface reference you can store any of its subclass object}.
- Interfaces **cannot be instantiated, they can only be implemented by classes or extended by other interfaces**.
- Java 7 interface can only contain **public abstract methods** and **static final fields (constants)**. They **cannot have non-static fields, non-static methods, and constructors**.
- Examples:
 - java.io.Closeable / java.io.AutoCloseable
 - java.lang.Runnable
 - java.util.Collection, java.util.List, java.util.Set.
- Interfaces can have **public static final fields**.

```
interface Shape {  
    /*public static final*/ double PI = 3.142;
```

```
/*public abstract*/ double calcArea();
/*public abstract*/ double calcPeri();
}
```

- Multiple interface inheritance is allowed.

```
class Person implements Acceptable, Displayable {
    ////....
}
```

- Interfaces can have public static final fields.

```
Interface : I1, I2, I3
Class : C1, C2, C3
class C1 implements I1 // okay
class C1 implements I1, I2 // okay
interface I2 implements I1 // error
interface I2 extends I1 // okay
interface I3 extends I1, I2 // okay
class C2 implements C1 // error
class C2 extends C1 // okay
class C3 extends C1, C2 // error
interface I1 extends C1 // error
interface I1 implements C1 // error
class C2 implements I1, I2 extends C1 // error
class C2 extends C1 implements I1,I2 // okay
```

class vs abstract class vs interface

- class
 - Has fields, constructors, and methods.
 - Can be used standalone -- create objects and invoke methods.
 - Reused in sub-classes -- inheritance.
 - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism.
- abstract class

- Has fields, constructors, and methods.
- Cannot be used independently -- can't create object.
- Reused in sub-classes -- inheritance -- Inherited into sub-class and "must override abstract methods".
- Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism.
- interface
 - Has only method declarations.
 - Cannot be used independently -- can't create object.
 - Doesn't contain anything for reusing (except static final fields).
 - Used as contract/specification -- Inherited into sub-class and "must override all methods".
 - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism.

Marker interfaces

Interface that doesn't contain any method declaration is called as "Marker interface".

- ***These interfaces are used to mark or tag certain functionalities/features in implemented class.*** In other words, they associate some information (metadata) with the class.
- ***Marker interfaces are used to check if a feature is enabled/allowed for the class.***
- Java has a few pre-defined marker interfaces. e.g. Serializable, Cloneable, etc.
- ***java.io.Serializable*** -- Allows JVM to convert object state into sequence of bytes.
- ***java.lang.Cloneable*** -- Allows JVM to create copy of the class object.

Cloneable interface

- ***Enable creating copy/clone of the object.***
- If a class is Cloneable, "Object.clone() method creates a shallow copy of the object". If class is not Cloneable, Object.clone() throws ***CloneNotSupportedException***.

- A class should implement Cloneable and **override clone() to create a deep/shallow copy of the object.**

```

class Date implements Cloneable {
    private int day, month, year;

    // ...
    // shallow copy
    public Object clone() throws CloneNotSupportedException {
        Date temp = (Date)super.clone();
        return temp;
    }

    //////////////////////////////////////

class Person implements Cloneable {
    private String name;
    private int weight;
    private Date birth;
    // ...
    // deep copy
    public Object clone() throws CloneNotSupportedException {
        Person temp = (Person)super.clone(); // shallow copy
        temp.birth = (Date)this.birth.clone(); // + copy reference types
        explicitly
        return temp;
    }
}

////////////////////////////////////

class Program {
    public static void main(String[] args) throws CloneNotSupportedException {
        Date d1 = new Date(28, 9, 1983);
        System.out.println("d1 = " + d1.toString());
        Date d2 = (Date)d1.clone();
        System.out.println("d2 = " + d2.toString());
        Person p1 = new Person("Nilesh", 70, d1);
        System.out.println("p1 = " + p1.toString());
        Person p2 = (Person)p1.clone();
        System.out.println("p2 = " + p2.toString());
    }
}

```

Multiple Inheritance is not supported through class in Java so to avoid certain challenges like **Ambiguity** and **diamond problems**.

Types of interfaces in Java are mentioned below:

1. Functional Interface
2. Marker interface

Interfaces "can be" used create ***immutable objects***,

1. create interface with only getter methods.

```
public interface ImmutablePoint2D {  
    public int getX();  
    public int getY();  
}
```

2. make your class implement above interface

```
public class Point2D implements ImmutablePoint2D{  
    private int x;  
    private int y;  
    public Point2D(int x, int y) { this.x = x; this.y = y; }  
  
    public int getX() { return this.x; }  
    public int getY() { return this.y; }  
  
    public void setX(int newX) { this.x = newX; }  
    public void setY(int newY) { this.y = newY; }  
}
```

3. Create a "reference of that Interface" and store "object of your class" in it.

```
ImmutablePoint2D point = new Point2D(0,0); // a concrete instance of Point2D is  
referenced by the immutable interface  
int x = point.getX(); // valid method call  
int y = point.setX(42); // compile error: the method setX() does not exist on type  
ImmutablePoint2D
```

4. If you call the setter method, it will give a compiler error because the reference is of interface type, and due to object slicing, it can only call getters and not setters.

Garbage collection

- Garbage collection is **automatic memory management by JVM**.
- **If a Java object is unreachable** (i.e. not accessible through any reference), **then it is automatically released by the garbage collector**.
- An object become eligible for GC in one of the following cases: **1. Nullify the reference**.
- **GC is a background thread in JVM that runs periodically and reclaim memory of unreferenced objects**.

```
MyClass obj = new MyClass();  
obj = null;
```

2. Reassign the reference.

```
MyClass obj = new MyClass();  
obj = new MyClass();
```

3. Object created locally in method.

```
void method() {  
    MyClass obj = new MyClass();  
    // ...  
}
```

4. Island of isolation i.e. objects are referencing each other, but not referenced externally.

```
class FirstClass {  
    private SecondClass second;  
    public void setSecond(SecondClass second) {  
        this.second = second;  
        second.setFirst(this);  
    }  
}  
class SecondClass {  
    private FirstClass first;  
    public void setFirst(FirstClass first) {  
        this.first = first;  
    }  
}  
class Main {  
    public static void method() {  
        FirstClass f = new FirstClass();
```

```

        f.setSecond(new SecondClass());
        f =
null;
    }
    // ...
}

```

- **Before object is destroyed, its `finalize()` method is invoked (if present).**
- One should override this method if object holds any resource to be released explicitly e.g. **file close**, **database connection**, etc.

```

class MyClass {
    private Connection con;
    public MyClass() throws Exception {
        con = DriverManager.getConnection("url", "username", "password");
    }
    // ...
    @Override
    public void finalize() {
        try {
            if(con != null)
                con.close();
        }
        catch(Exception e) {
        }
    }
}

class Main {
    public static void method() throws Exception {
        MyClass my = new MyClass();
        my =
null;
        System.gc(); // request GC
    }
    // ...
}

```

- GC can be requested (not forced) by one of the following.
 - 1. `System.gc();`**
 - 2. `Runtime.getRuntime().gc();`**

```

public static void main(String[] args) {

    Runtime rt = Runtime.getRuntime();
    rt.gc(); // request garbage collector
    //System.gc(); // internally --> Runtime.getRuntime().gc();
}

```

- GC is of two types i.e. Minor and Major.
 - **Minor GC**: Unreferenced objects from **young generation** are reclaimed. Objects not reclaimed here are moved to old/permanent generation.
 - **Major GC**: Unreferenced objects from **all generations** are reclaimed. This is unefficient (slower process).
- JVM GC internally use **Mark and Compact algorithm**.

Resource Management

- System resources should be released immediately after the use. Few system resources are Memory, File, IO Devices, Socket/Connection, etc.
- The Garbage collector automatically releases memory if objects are no more used (unreferenced).
- The GC collector doesn't release memory/resources immediately; rather it is executed only memory is full upto a threshold.
- **The standard way to release the resources immediately after their use is `java.io.Closeable` interface.** It has **only one method**.
 - void **close()** throws IOException;
- *Programmer should call close() **explicitly** on resource object after its use.*
 - e.g. FileInputStream, FileOutputStream, etc.
- Java 7 introduced an interface `java.lang.AutoCloseable` as super interface of `Closeable`. It has only one method.
 - void close() throws Exception;
- Since it is super-interface of `Closeable`, all classes implementing `Closeable` now also inherit from `AutoCloseable`.
- **If a class is inherited from `AutoCloseable`, then it can be closed using **try-with-resource syntax**.**

```
class MyResource implements AutoCloseable {  
    // ...  
    public void close() {  
        // cleanup code  
    }  
}  
  
class Program {
```



```
public static void main(String[] args) {  
    try(MyResource res = new MyResource()) {  
        // ...  
    }  
    // res.close() called automatically  
}  
}
```

- The *Scanner class is also AutoCloseable*.

```
class Program {  
    public static void main(String[] args) {  
        try(Scanner sc = new Scanner(System.in)) {  
            // ...  
        }  
        // sc.close() is auto-closed  
    }  
}
```

Exception Handling

- ***An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.***
- If these ***problems cannot be handled in the current method***, then they should be ***sent back to the calling method***.
- Java keywords for exception handling
 - throw
 - try
 - catch
 - finally
 - throws

```
static double divide(int numerator, int denominator) {  
    if(denominator == 0)  
        throw new RuntimeException("Cannot divide by zero");  
    return (double)numerator / denominator;  
}  
public static void main(String[] args) {  
    // ...  
}
```

```

try
{
    int num1 = sc.nextInt();
    int num2 = sc.nextInt();
    double result = divide(num1, num2);
    System.out.println("Result: " + result);
}
catch(RuntimeException e)
{
    e.printStackTrace();
}
}

```

- Java operators/APIs throw **pre-defined exception** if runtime problem occurs.
- For example, **ArithmeticException** is thrown when divide by zero is tried.

Java exception class hierarchy

```

Object
|- Throwable
|   |- Error
|   |   |- AssertionError
|   |   |- VirtualMachineError
|   |   |   |- StackOverflowError
|   |   |   |- OutOfMemoryError
|   |- Exception
|       |- CloneNotSupportedException
|       |- IOException
|       |   |- EOFException
|       |   |- FileNotFoundException
|       |- SQLException
|       |- InterruptedException
|       |- RuntimeException
|           |- NullPointerException
|           |- ArithmeticException
|           |- NoSuchElementException
|           |   |- InputMismatchException
|       |- IndexOutOfBoundsException
|           |- ArrayIndexOutOfBoundsException
|           |- StringIndexOutOfBoundsException

```

- One catch block cannot handle problems from multiple try blocks.
- One try block may have multiple catch blocks. Specialized catch block must be written **before** generic catch block.
- **If certain code to be executed irrespective of exception occur or not, write it in "finally block".**

```
try {  
    // file read code -- possible problems  
    // 1. file not found  
    // 2. end of file is reached  
    // 3. error while reading from file  
    // 4. null reference (programmer's mistake)  
}  
catch(NullPointerException ex) {  
    // ...  
}  
catch(FileNotFoundException ex) {  
    // ...  
}  
catch(EOFException ex) {  
    // ...  
}  
catch(IOException ex) {  
    // ...  
}  
finally {  
    // close the file  
}
```

- When exception is raised, it will be caught by nearest matching catch block. ***If no matching catch block is found, the exception will be caught by JVM and it will abort the program.***

java.lang.Throwable class

- Throwable is ***root class for all errors and exceptions*** in Java.
- ***Only objects that are one of the subclasses of this class are thrown by any "Java Virtual Machine" or may be thrown by the Java throw statement.***
- Methods
 - Throwable()
 - Throwable(String message)
 - Throwable(Throwable cause)
 - Throwable(String message, Throwable cause)
 - String getMessage()
 - void printStackTrace()
 - void printStackTrace(PrintStream s)
 - void printStackTrace(PrintWriter s)

- String toString()
- Throwable getCause()

java.lang.Error class

- **Usually represents the runtime problems that are not recoverable.**
- Generated due to environmental condition/Runtime environment (**e.g. OS error, Memory error, etc.**)
- Examples:
 - AssertionError
 - VirtualMachineError
 - StackOverflowError
 - OutOfMemoryError

java.lang.Exception class

- **Represents the runtime problems that can be handled.**
- The exception is handled using try-catch block.
- Examples:
 - CloneNotSupportedException
 - IOException
 - SQLException
 - NullPointerException
 - ArrayIndexOutOfBoundsException
 - ClassCastException.

Exception types

- There are two types of exceptions
 - **Checked exception -- Checked by compiler and forced to handle.**
 - Unchecked exception -- Not checked by compiler.

Unchecked exception

- **RuntimeException** and **all its sub classes** are unchecked exceptions.
- **Typically represents programmer's or user's mistake.**

- `NullPointerException`, `NumberFormatException`, `ClassCastException`, etc.
- Compiler doesn't provide any checks -- if exception is handled or not.
- Programmer may or may not handle (catch block) the exception. If exception is not handled, it will be caught by JVM and abort the application.

Checked exception

- **Exception and all its sub classes (except `RuntimeException`)** are checked exceptions.
- Typically represents **problems arised out of JVM/Java** i.e. at OS/System level.
 - `IOException`, `SQLException`, `InterruptedException`, etc.
- **Compiler checks if the exception is handled** in one of following ways.
 1. Matching catch block to handle the exception.

```
void someMethod() {  
    try {  
        // file io code  
    }  
    catch(IOException ex) {  
        // ...  
    }  
}
```

2. **throws clause** indicating exception to be handled by calling method.

```
void someMethod() throws IOException {  
    // file io code  
}  
void callingMethod() {  
    try {  
        someMethod();  
    }  
    catch(IOException ex) {  
        // ...  
    }  
}
```

Exception handling keywords

- "try" block
 - Code where runtime problems may arise should be written in try block.
 - try block must have one of the following
 - catch block
 - finally block
 - try-with-resource
 - Can be nested in try, catch, or finally block.
- "throw" statement
 - Throws an exception/error i.e. any object that is inherited from Throwable class.
 - Can throw only one exception at time.
 - All next statements are skipped and control jumps to matching catch block.
- "catch" block
 - Code to handle error/exception should be written in catch block.
 - Argument of catch block must be Throwable or its sub-class.
 - Generic catch block -- Performs upcasting -- Should be last (if multiple catch blocks)

```
try {  
    // ...  
}  
catch(Throwable e) {  
    // can handle exception of any type  
}
```

- Multi-catch block -- Same logic to execute different exceptions.

```
try {  
    // ...  
}  
catch(ArithmeticException | InputMismatchException e) {  
    // common logic to handle ArithmeticException and InputMismatchException  
}
```

- Exception sub-class should be caught before super-class.

```
try {  
    // ...  
}
```

```
catch EOFException ex) {  
    // ...  
}  
catch IOException ex) {  
    // ...  
}
```

- "finally" block
 - Resources are closed in finally block.
 - Executed irrespective of exception occurred or not.
 - finally block not executed only if JVM/application exits (System.exit()).

```
Scanner sc = new Scanner(System.in);  
try {  
    // ...  
}  
catch (Exception ex) {  
    // ...  
}  
finally {  
    sc.close();  
}
```

- "throws" clause
 - **Written after method declaration to specify list of exception not handled by called method and to be handled by calling method.**
 - **Writing unhandled checked exceptions in throws clause is compulsory.**
 - The **unchecked exceptions** written in **throws clause** are **ignored by the compiler.**

```
void someMethod() throws IOException, SQLException {  
    // ...  
}
```

- **Sub-class overridden method can throw same or subset of exception from super-class method.**

```
class SuperClass {  
    // ...  
    void method() throws IOException, SQLException, InterruptedException {  
        }  
    }  
class SubClass extends SuperClass {  
    // ...  
    void method() throws IOException, SQLException {  
        }  
    }
```

```
}  
}
```

```
class SuperClass {  
    // ...  
    void method() throws IOException {  
        }  
    }  
    class SubClass extends SuperClass {  
        // ...  
        void method() throws FileNotFoundException, EOFException {  
            }  
        }  
    }
```

Exception chaining

- Sometimes an exception is generated due to another exception. For example, database `SQLException` may be caused due to network problem `SocketException`.
- To represent this an exception can be chained/nested into another exception.
- ***If method's throws clause doesn't allow throwing exception of certain type, it can be nested into another (allowed) type and thrown.***

```
class DatabaseServlet extends HttpServlet {  
  
    @Override  
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws  
        IOException, ServletException  
    {  
        try {  
            // JDBC code throwing SQLException  
        }  
        catch(SQLException ex) {  
            ex.printStackTrace();  
            //throw ex; // to the Java web server -- cannot throw SQLException - throws clause  
            throw new ServletException(ex); // to the Java web server  
        }  
    }  
}
```



```
public static int divide(int num, int den) throws Exception {
    try {
        return num / den;
    }
    catch(ArithmeticException ex) {
        throw new Exception(ex);
    }
}

public static void main(String[] args) {
    try {
        int result = divide(22, 0);
        System.out.println(result);
    }
    catch(Exception ex) {
        ArithmeticException e = (ArithmeticException) ex.getCause();
        System.out.println(e.getMessage());
    }
}
```

- Sometimes an exception raised is not allowed to throw (due to throws clause). In such cases, raised exception is wrapped in some allowed exception type and thrown. This is exception chaining.

User defined exception class

- ***If pre-defined exception class are not suitable to represent application specific problem, then user-defined exception class should be created.***
- User defined exception class may contain fields to store additional information about problem and methods to operate on them.
- ***Typically exception class's constructor call super class constructor to set fields like message and cause.***
- ***If class is inherited from RuntimeException, it is used as unchecked exception. If it is inherited from Exception, it is used as checked exception.***

```
class InvalidTimeException extends RuntimeException {

    private String field; // seconds, minutes, hours
    private int value; // wrong value

    public InvalidTimeException() {
        super("Invalid time");
    }
}
```

```

public InvalidTimeException(String field, int value) {
    super("Invalid " + field + ": " + value);
    this.field = field;
    this.value = value;
}

public String getField() {
    return field;
}

public int getValue() {
    return value;
}
// ...
}

////////////////////////////////////
class Time {
    public void setMin(int min) {
        if(min <= 0 || min >= 60)
            throw new InvalidTimeException("minutes", min);
        this.min = min;
    }
}

```

The screenshot shows a code editor with a sidebar on the left displaying a file tree. The main editor area shows a Java exception hierarchy tree. Red arrows point from handwritten notes to specific parts of the hierarchy.

Object

- Throwable** (general runtime problems -- getMessage(), printStackTrace(), ex chaining.)
 - Error** (runtime problems usually from which cannot be recovered -- system level problems)
 - AssertionError
 - VirtualMachineError
 - StackOverflowError
 - OutOfMemoryError
 - Exception** (runtime problems that needs to be and can be handled -- take action in catch block.)
 - CloneNotSupportedException
 - IOException
 - EOFException
 - FileNotFoundException
 - SQLException
 - InterruptedException
 - RuntimeException** (checked exceptions - usually represents runtime problems which may need handling outside to JVM (i.e. at OS level). Java compiler force programmer to handle these exceptions -- either try-catch block or use throws clause.)
 - NullPointerException
 - ArithmeticException
 - NoSuchElementException
 - InputMismatchException
 - IndexOutOfBoundsException
 - ArrayIndexOutOfBoundsException
 - StringIndexOutOfBoundsException

unchecked exceptions - usually represents runtime problems which occur due to user's or programmer's mistake. Java compiler do not check whether exception is handled or not. If any exception is unhandled, it will be caught by JVM and will terminate the appln.

Java Strings

- java.lang.Character is wrapper class that represents char. In Java, each char is 2 bytes because it follows unicode encoding.
- String is **sequence of characters**.
 1. **java.lang.String: "Immutable" character sequence.**
 2. **java.lang.StringBuffer: Mutable character sequence (Thread-safe).**
 3. **java.lang.StringBuilder: Mutable character sequence (Not Thread-safe).**
- String helpers
 1. **java.util.StringTokenizer: Helper class to split strings.**

String objects

- java.lang.String is class and **strings in java are "objects"**.
- String constants/literals are stored in **string pool**.

```
String str1 = "Sunbeam";  
  
str1 ==> object;  
"Sunbeam" ==> String Literal stored in string pool.
```

- String objects created using "new" operator are allocated on heap.

```
String str2 = new String("Nilesh");
```

- In java, **String is immutable**. **If try to modify, it creates a new String object on heap.**

String literals

- Since strings are immutable, **string constants are not allocated multiple times**.
- **String constants/literals are stored in string pool. Multiple references may refer the same object in the pool.**
- String pool is also called as String literal pool or String constant pool.
- From Java 7, String pool is in the **heap space (of JVM)**.

- **The string literal objects are created during "class loading".**

String objects vs String literals

- Important points
 - **Two strings with same contents are never repeated in String pool.**
 - `==` operator **compares addresses/references** of two objects.
 - **`equals()` compares contents** of two String objects -- case sensitive.
 - **`intern()`** adds a string in string pool (if not already present) and return its reference.
- Example 01:

```
String s1 = "Sunbeam";
String s2 = "Sunbeam";
System.out.println(s1 == s2);      // true
System.out.println(s1.equals(s2)); // true
```

- Example 02:

```
String s1 = new String("Sunbeam");
String s2 = new String("Sunbeam");
System.out.println(s1 == s2);      // false
System.out.println(s1.equals(s2)); // true
```

- Example 03:

```
String s1 = "Sunbeam";
String s2 = new String("Sunbeam");
System.out.println(s1 == s2);      // false
System.out.println(s1.equals(s2)); // true
```

- Example 04:

```
String s1 = "Sunbeam";
String s2 = "Sun" + "beam";
System.out.println(s1 == s2);      // true
System.out.println(s1.equals(s2)); // true
```

- Example 05:

```
String s1 = "Sunbeam";
String s2 = "Sun";
String s3 = s2 + "beam";
System.out.println(s1 == s3);      // false
System.out.println(s1.equals(s3)); // true
```

- Example 06:

```
String s1 = "Sunbeam";
String s2 = new String("Sunbeam").intern();
System.out.println(s1 == s2);      // true
System.out.println(s1.equals(s2)); // true
```

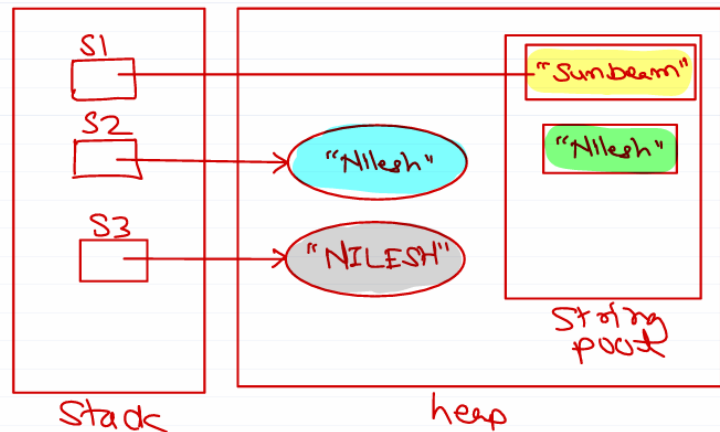
- Example 07:

```
String s1 = "Sunbeam";
String s2 = "SunBeam";
System.out.println(s1 == s2);
// false
System.out.println(s1.equals(s2)); // false
System.out.println(s1.equalsIgnoreCase(s2)); // true
System.out.println(s1.compareTo(s2)); // 32
System.out.println(s1.compareToIgnoreCase(s2)); // 0
```

```
String str1 = "Ramdeo";
String str2 = "Ramraj";
int diff = str1.compareTo(str2);
System.out.println("Diff: " + diff); // 'd' - 'r'
```

Strings

```
String s1 = "Sunbeam";
String s2 = new String("Nilesh");
String s3 = s2.toUpperCase();
```



Q & A

1. Why Strings in Java are immutable?

- Java has String pool to reduce memory usage. String doesn't store same string multiple times i.e. only one copy of the one string kept. This can be implemented only if strings are immutable.
- In java type/class names, method name, field names are internally stored as strings (utf-8). Any change done in these strings at runtime will cause unexpected results. This can be a security threat.
- To avoid these problems, Java strings are immutable.

2. Does StringBuilder use String pool?

- No. Only string literal will be stored in pool, but StringBuilder object will be created in heap.
- StringBuilder sb = new StringBuilder("Sunbeam");

4. String concat vs StringBuilder.

- String concatenation internally creates new string objects (for each concat). This is unefficient (time and space).
- StringBuilder is efficient option for building strings with multiple concat (append).

```
String str1 = "Sun";
String str2 = "Beam"
String str3 = "DAC";
String s1 = str1 + str2 + str3;
// s1 = str1.concat(str2).concat(str3);
StringBuilder sb = new StringBuilder();
String s2 = sb.append(str1).append(str2).append(str3).toString();
// ...
```

Prepared by: Nilesh Ghule 2 / 13

- Why StringBuilder/StringBuffer equals() returns false even if contents are same?
 - equals() method is not overridden in StringBuffer/StringBuilder class.
 - So equals() from Object class is invoked. It compares addresses/references.
 - Now even if two StringBuffer/StringBuilder objects have same contents, their address will differ and equals() will return false.
- substr = str.substring(6, 1);
 - startIndex=6, endIndex=1 -> cause IndexOutOfBoundsException.

String operations

- int length()
- char charAt(int index)

- `int compareTo(String anotherString)`
- `boolean equals(String anotherString)`
- `boolean equalsIgnoreCase(String anotherString)`
- `boolean matches(String regex)`
- `boolean isEmpty()`
- `boolean startsWith(String prefix)`
- `boolean endsWith(String suffix)`
- `int indexOf(int ch)`
- `int indexOf(String str)`
- `String concat(String str)`
- `String substring(int beginIndex)`
- `String substring(int beginIndex, int endIndex)`
- `String[] split(String regex)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String trim()`
- `byte[] getBytes()`
- `char[] toCharArray()`
- `String intern()`
- `static String valueOf(Object obj)`
- `static String format(String format, Object... args)`

StringBuffer vs StringBuilder

- `StringBuffer` and `StringBuilder` are final classes declared in `java.lang` package.
- It is ***used create to "mutable" string instance.***
- ***`equals()` and `hashCode()` method is "not" overridden inside it.***
- Can create instances of these classes using ***new operator only***. Objects are created on heap.
- `StringBuffer` capacity grows if size of internal char array is less (than string to be stored).
 - The default capacity is **16**.

```
int max = (minimumCapacity > value.length? value.length * 2 + 2 : value.length);
minimumCapacity = (minimumCapacity < max? max : minimumCapacity);
char[] nb = new char[minimumCapacity];
```

- `StringBuffer` implementation is thread safe (slower execution) while `StringBuilder` is not thread-safe (faster execution).
- `StringBuilder` is introduced in Java 5.0 for better performance in single threaded applications.

- Example 01:

```
StringBuffer s1 = new StringBuffer("Sunbeam");
StringBuffer s2 = new StringBuffer("Sunbeam");
System.out.println(s1 == s2);
// ???
System.out.println(s1.equals(s2)); // ???
```

- Example 02:

```
StringBuffer s1 = new StringBuffer("Sunbeam");
String s2 = new String("Sunbeam");
System.out.println(s1 == s2);
// ???
System.out.println(s1.equals(s2)); // ???
```

- Example 03:

```
String s1 = new String("Sunbeam");
StringBuffer s2 = new StringBuffer("Sunbeam");
System.out.println(s1.equals(s2)); // ???
System.out.println(s1.equals(s2.toString())); // ???
```

- Example 04:

```
StringBuffer s1 = new StringBuffer("Sunbeam");
StringBuffer s2 = s1.reverse();
System.out.println(s1 == s2); // ???
System.out.println(s1.equals(s2)); // ???
```

- Example 05:

```
StringBuilder s1 = new StringBuilder("Sunbeam");
StringBuilder s2 = new StringBuilder("Sunbeam");
System.out.println(s1 == s2); // ???
System.out.println(s1.equals(s2)); // ???
```

- Example 06:

```
StringBuffer s = new StringBuffer();
System.out.println("Capacity: " + s.capacity() + ", Length: " + s.length()); //
16, 0
```



```
s.append("1234567890");
System.out.println("Capacity: " + s.capacity() + ", Length: " + s.length()); //
16, 10
s.append("ABCDEFGHJKLMNOPQRSTUVWXYZ");
System.out.println("Capacity: " + s.capacity() + ", Length: " + s.length()); //
34, 32
```

StringTokenizer

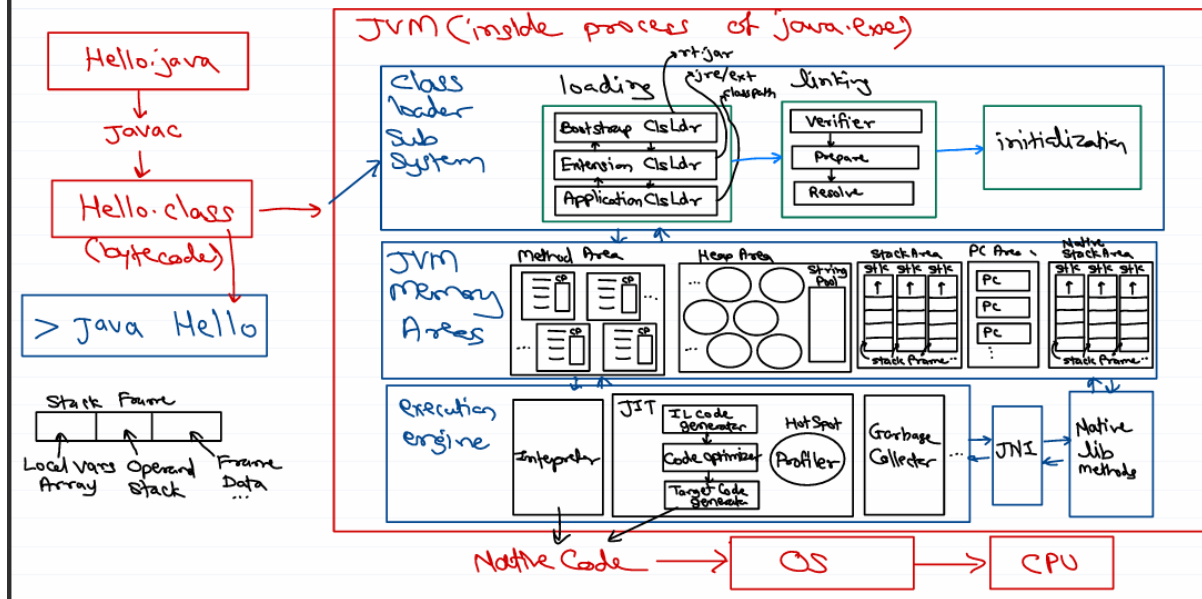
- **Used to break a string into multiple tokens** - like split() method.
- Methods of java.util.StringTokenizer
 - int countTokens()
 - boolean hasMoreTokens()
 - String nextToken()
 - String nextToken(String delim)

```
String str = "My name is Bond, James Bond.";
String delim = " ,.";
StringTokenizer tokenizer = new StringTokenizer(str, delim);
while(tokenizer.hasMoreTokens()) {
    String token = tokenizer.nextToken();
    System.out.println(token);
}
```

Refer DAY 09 => day09.pdf

JVM Architecture

JVM architecture



Java compilation process

- `Hello.java` --> Java Compiler --> `Hello.class`
 - `javac Hello.java`
- Java compiler converts Java code into the Byte code.

Byte code

- Byte code is machine level instructions that can be executed by Java Virtual Machine (JVM).
 - Instruction = Op code + Operands
 - e.g. `iadd op1, op2`
- Each Instruction in byte-code is of 1 byte.
 - `.class` --> JVM --> Windows (x86)
 - `.class` --> JVM --> Linux (ARM)
- JVM converts byte-code into target machine/native code (as per architecture).

.class format

- .class file contains header, byte-code, meta-data, constant-pool, etc.
- .class header contains
 - magic number -- `0xCAFEBADE` (first 4 bytes of .class file)
 - information of other sections.
- .class file can inspected using "javap" tool.
 - terminal> `javap java.lang.Object`
 - Shows public and protected members.
 - terminal> `javap -p java.lang.Object`

- shows private members as well.
- terminal> javap -c java.lang.Object
 - Shows byte-code of all methods
- terminal> javap -v java.lang.Object
 - Detailed (verbose) information in .class
 - Constant pool
 - Methods & their byte-code
 - ...
- "javap" tool is part of JDK.

Executing Java program (.class)

- terminal> java Hello
- "java" is a Java Application Launcher.
- java.exe (disk) --> Loader --> (Windows OS) Process.
- When "java" process executes, JVM (jvm.dll) gets loaded in the process.
- JVM will now find (in CLASSPATH) and execute the .class.

JVM Architecture (Overview)

Loading

Linking

Initialization

JVM memory areas

Method area

Heap area

Stack area

PC Registers

Native method stack area

- Separate native method stack is created for each thread in JVM (when thread is created).
- When a native method is called from the stack, a stack frame is created on its stack.

Execution engine

enum

- "enum" keyword is added in Java 5.0.
- ***Used to make constants to make code more readable.***
- The switch constants can be made more readable using Java enums.

```
enum ArithmeticOperations {  
    ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION;  
}  
  
ArithmeticOperations choice = ArithmeticOperations.ADDITION;
```

```
// ...
switch(choice) {
case ADDITION:
    c = a + b;
break;
case SUBTRACTION:
    c = a - b;
break;
// ...
}
```

- In java, enums **cannot be declared locally** (within a method).
- The **declared enum is converted into enum class**.

```
// user-defined enum
enum ArithmeticOperations {
    ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION
}
```

```
// generated enum code
final class ArithmeticOperations extends Enum {
public static ArithmeticOperations[] values() {
return (ArithmeticOperations[])$VALUES.clone();
}
public static ArithmeticOperations valueOf(String s) {
return (ArithmeticOperations)Enum.valueOf(ArithmeticOperations, s);
}
private ArithmeticOperations(String name, int ordinal) {
super(name, ordinal); // invoke sole constructor Enum(String,int);
}
public static final ArithmeticOperations ADDITION;
public static final ArithmeticOperations SUBTRACTION;
public static final ArithmeticOperations MULTIPLICATION;
public static final ArithmeticOperations DIVISION;
private static final ArithmeticOperations $VALUES[];
static {
    ADDITION = new ArithmeticOperations("ADDITION", 0);
    SUBTRACTION = new ArithmeticOperations("SUBTRACTION", 1);
    MULTIPLICATION = new ArithmeticOperations("MULTIPLICATION", 2);
    DIVISION = new ArithmeticOperations("DIVISION", 3);
    $VALUES = (new ArithmeticOperations[] {
        ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION
    });
}
}
```

- The enum type declared is **implicitly inherited from java.lang.Enum class**. So it cannot be extended from another class, but enum may implement interfaces.

- The enum constants declared in enum are *public static final* fields of generated class.
- **Enum objects cannot be created explicitly (as generated constructor is private).**
- The generated class will have a **"values()"** method that **returns array of all constants** and **"valueOf()" method to convert String to enum constant.**
- The enums constants can be used in switch-case and can also be compared using == operator.
- The enum may have fields and methods.

```
enum Element {
    H(1, "Hydrogen"),
    HE(2, "Helium"),
    LI(3, "Lithium");
private final int atomNum;
private final String atomName;
private Element(int atomNum, String atomName) {
    this.atomNum = atomNum;
    this.atomName = atomName;
}
public int getAtomNum() {
    return atomNum;
}
public String getAtomName() {
    return atomName;
}
// ...
}
```

The java.lang.Enum class has following members:

```
public abstract class Enum<E> implements java.lang.Comparable<E>,
java.io.Serializable {
    private final String name;
    private final int ordinal;

    protected Enum(String,int); // sole constructor - can be called from user-defined
enum class only
    public final String name(); // name of enum const
    public final int ordinal(); // position of enum const (0-based)

    public String toString(); // returns name of const
    public final int compareTo(E); // compares with another enum of same type on basis
of ordinal number
    public static <T> T valueOf(Class<T>, String);
    // ...
}
```

Generic Programming

- Code is said to be generic if same code can be used for various (practically all) types.
- Best example:
 - Data structure e.g. Stack, Queue, Linked List, ...
 - Algorithms e.g. Sorting, Searching, ...
- Two ways to do Generic Programming in Java
 - using `java.lang.Object` class -- Non typesafe -- Till Java 1.4
 - using Generics -- Typesafe -- Added in Java 5.0

Generic Programming Using `java.lang.Object`

```
class Box {  
    private Object obj;  
    public void set(Object obj) {  
        this.obj = obj;  
    }  
    public Object get() {  
        return this.obj;  
    }  
}
```

```
Box b1 = new Box();  
b1.set("Nilesh");  
String obj1 = (String)b1.get();  
System.out.println("obj1 : " + obj1);  
Box b2 = new Box();  
b2.set(new Date());  
Date obj2 = (Date)b2.get();  
System.out.println("obj2 : " + obj2);  
Box b3 = new Box();  
b3.set(new Integer(11)); String obj3 = (String)b3.get(); // ??  
System.out.println("obj3 : " + obj3);
```

Generic Programming Using Generics

- Added in Java 5.0.
- Similar to templates in C++.
- We can implement
 - **Generic classes**
 - **Generic methods**
 - **Generic interfaces.**

Advantages of Generics

- **Stronger type checking at compile time i.e. type-safe coding.**
- **Explicit type casting is not required.**
- **Generic data structure and algorithm implementation.**

Generic Classes

- Implementing a generic class

```
class Box<TYPE> {  
    private TYPE obj;  
    public void set(TYPE obj) {  
        this.obj = obj;  
    }  
    public TYPE get() {  
        return this.obj;  
    }  
}
```

```
Box<String> b1 = new Box<String>();  
b1.set("Nilesh");  
String obj1 = b1.get();  
System.out.println("obj1 : " + obj1);  
Box<Date> b2 = new Box<Date>();  
b2.set(new Date());  
Date obj2 = b2.get();  
System.out.println("obj2 : " + obj2);  
Box<Integer> b3 = new Box<Integer>();  
b3.set(new Integer(11));  
String obj3 = b3.get(); // ??  
System.out.println("obj3 : " + obj3);
```

- Instantiating generic class


```
Box<String> b1 = new Box<String>(); // okay
Box<String> b2 = new Box<>(); // okay -- type inference
Box<> b3 = new Box<>(); // compiler error -- type must be given while creating
generic class reference
// cannot be auto-detected
Box<Object> b4 = new Box<String>(); // compiler error
Box b5 = new Box(); // okay -- internally considered "Object" type -- compiler
warning "raw types"
Box<Object> b6 = new Box<Object>(); // okay -- Not usually required/used
```

T is placeholder for datatype.s

- Generic types naming convention
 1. T : Type
 2. N : Number
 3. E : Element
 4. K : Key
 5. V : Value
 6. S,U,R : Additional type param

Bounded generic types

- Bounded generic param "***restricts data" type that can be used as type argument.***
- Decided by the developer of the generic class.

```
class Box<T extends Number> {
    private T obj;
    public T get() {
        return this.obj;
    }
    public void set(T obj) {
        this.obj = obj;
    }
}
```

- The Box<> can now be used only for the classes inherited from the Number class.

```
Box<Number> b1 = new Box<>(); // okay
Box<Boolean> b2 = new Box<>(); // error
Box<Character> b3 = new Box<>(); // error
Box<String> b4 = new Box<>(); // error
Box<Integer> b5 = new Box<>(); // okay
Box<Double> b6 = new Box<>(); // okay
Box<Date> b7 = new Box<>(); // error
Box<Object> b8 = new Box<>(); // error
```

Unbounded generic types

- Unbounded generic type is indicated with wild-card "?".
- Can be given while declaring generic class reference.

```
class Box<T> {  
    private T obj;  
    public Box(T obj) {  
        this.obj = obj;  
    }  
    public T get() {  
        return this.obj;  
    }  
    public void set(T obj) {  
        this.obj = obj;  
    }  
}
```

```
public static void printBox(Box<?> b) {  
    Object obj = b.get();  
    System.out.println("Box contains: " + obj);  
}
```

```
Box<Number> nb = new Box<Number>(123.45);  
printBox(nb); // okay  
Box<Integer> ib = new Box<Integer>(100);  
printBox(ib); // okay  
Box<Object> ob = new Box<Object>("test");  
printBox(ob); // okay  
Box<Date> db = new Box<Date>(new Date());  
printBox(db); // okay
```

Upper bounded generic types

- Generic param type can be the **given class or its "sub"-class**.

```
public static void printBox(Box<? extends Number> b) {  
    Object obj = b.get();  
    System.out.println("Box contains: " + obj);  
}
```

```
Box<Number> nb = new Box<Number>(123.45);  
printBox(nb); // okay  
Box<Integer> ib = new Box<Integer>(100);  
printBox(ib); // okay  
Box<Object> ob = new Box<Object>("test");  
printBox(ob); // error  
Box<Date> db = new Box<Date>(new Date());  
printBox(db); // error
```

Lower bounded generic types

- Generic param type can be the **given class or its "super"-class**.

```
public static void printBox(Box<? super Number> b) {  
    Object obj = b.get();  
    System.out.println("Box contains: " + obj);  
}
```

```
Box<Number> nb = new Box<Number>(123.45);  
printBox(nb); // okay  
Box<Integer> ib = new Box<Integer>(100);  
printBox(ib); // error  
Box<Object> ob = new Box<Object>("test");  
printBox(ob); // okay  
Box<Date> db = new Box<Date>(new Date());  
printBox(db); // error
```

Generic Methods

- Generic methods are used to implement **generic algorithms**.

```
// non type-safe
void printArray(Object[] arr) {
    for(Object ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
// type-safe
<T> void printArray(T[] arr) {
    for(T ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
String[] arr1 = { "John", "Dagny", "Alex" };
printArray(arr1); // printArray<String> -- String type is inferred.
Integer[] arr2 = { 10, 20, 30 };
printArray(arr2); // printArray<Integer> -- Integer type is inferred.
```

Generics Limitations

1. **Cannot instantiate generic types with primitive Types.** Only reference types are allowed.

```
ArrayList<Integer> list = new ArrayList<Integer>(); // okay
ArrayList<int> list = new ArrayList<int>(); // compiler error
```

2. **Cannot create instances of Type parameters.**

3. **Cannot declare static fields/methods with generic type parameters.**

```
class Box<T> {
    private T obj; // okay
    private static T object; // compiler error
    // ...
}
```

4. **Cannot Use casts or instanceof with generic Type params.**

```
if(obj instanceof T) { // compiler error
    newobj = (T)obj;
// compiler error
}
```

5. **Cannot Create arrays of generic parameterized Types**

```
T[] arr = new T[5]; // compiler error
```

6. **Cannot create, catch, or throw Objects of Parameterized Types**

```
throw new T(); // compiler error
try {
// ...
} catch(T ex) { // compiler error
// ...
}
```

7. **Cannot overload a method just by changing generic type. Because after erasing/removing the type param, if params of two methods are same, then it is not allowed.**

```
public void printBox(Box<Integer> b) {
// ...
}
public void printBox(Box<String> b) { // compiler error
// ...
}
```

Type erasure

- **The generic type information is erased (not maintained) at runtime (in JVM).** Box and Box both are internally (JVM level) treated as Box objects. The field "T obj" in Box class, is treated as "Object obj".
- Because of this method overloading with generic type difference is not allowed.

```
void printBox(Box<Integer> b) { ... }
// void printBox(Box b) { ... } <-- In JVM
```

```
void printBox(Box<Double> b) { ... } //compiler error
// void printBox(Box b) { ... } <-- In JVM
```

Generic Interfaces

- Interface is standard/specification.

```
// Comparable is pre-defined interface -- non-generic till Java 1.4
interface Comparable {
    int compareTo(Object obj);
}

class Person implements Comparable {
    // ...
    public int compareTo(Object obj) {
        Person other = (Person)obj; // down-casting
        // compare "this" with "other" and return difference
    }
}

class Program {
    public static void main(String[] args) {
        Person p1 = new Person("James Bond", 50);
        Person p2 = new Person("Ironman", 45);
        int diff = p1.compareTo(p2);
        if(diff == 0)
            System.out.println("Both are same");
        else if(diff > 0)
            System.out.println("p1 is greater than p2");
        else //if(diff < 0)
            System.out.println("p1 is less than p2");
        01/11/2023
        diff = p2.compareTo("Superman"); // will fail at runtime with
        ClassCastException (in down-casting)
    }
}
```

- Generic interface has type-safe methods (arguments and/or return-type).

```
// Comparable is pre-defined interface -- generic since Java 5.0
interface Comparable<T> {
    int compareTo(T obj);
}

class Person implements Comparable<Person> {
    // ...
    public int compareTo(Person other) {
```

```
// compare "this" with "other" and return difference
    }
}
class Program {
public static void main(String[] args) {
    Person p1 = new Person("James Bond", 50);
    Person p2 = new Person("Ironman", 45);
    int diff = p1.compareTo(p2);
    if(diff == 0)
        System.out.println("Both are same");
    else if(diff > 0)
        System.out.println("p1 is greater than p2");
    else //if(diff < 0)
        System.out.println("p1 is less than p2");
        diff = p2.compareTo("Superman"); // compiler error
    }
}
```

Comparable<>

- **Standard for comparing the "current" object to the other object.**
- Also referred as "**Natural Ordering**" for the class.
- Has **single abstract method `int compareTo(T other);`**
- In **java.lang** package.
- Used by various methods like **`Arrays.sort(Object[])`**, ...

```
// pre-defined interface
interface Comparable<T> {
    int compareTo(T other);
}

class Employee implements Comparable<Employee> {
    private int empno;
    private String name;
    private int salary;
    // ...
    public int compareTo(Employee other) {
        int diff = this.empno - other.empno;
        return diff;
    }
}
```

```
Employee e1 = new Employee(1, "Sarang", 50000);
Employee e2 = new Employee(2, "Nitin", 40000);
int diff = e1.compareTo(e2);
```

```
Employee[] arr = { ... };
Arrays.sort(arr);
for(Employee e:arr)
    System.out.println(e);
```

Comparator<>

- *Standard for comparing two (other) objects.*
- Has single abstract method `int compare(T obj1, T obj2);`
- In **java.util** package.
- Used by various methods like **`Arrays.sort(T[], comparator)`**, ...

```
// pre-defined interface
interface Comparator<T> {
    int compare(T obj1, T obj2);
}
```

```
class EmployeeSalaryComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        if(e1.getSalary() == e2.getSalary())
            return 0;
        if(e1.getSalary() > e2.getSalary())
            return +1;
        return -1;
    }
}
```

Multi-level sorting


```
class Employee implements Comparable<Employee> {
    private int empno;
    private String name;
    private String designation;
    private int department;
    private int salary;
    // ...
}
```

```
// Multi-level sorting -- 1st level: department, 2nd level: designation, 3rd
level: salary(int)
class CustomComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        int diff = e1.getDepartment().compareTo(e2.getDepartment());
        if(diff == 0)
            diff = e1.getDesignation().compareTo(e2.getDesignation());
        if(diff == 0)
            diff = e1.getSalary() - e2.getSalary();
        return diff;
    }
}
```

```
Employee[] arr = { ... };
Arrays.sort(arr, new CustomComparator());
// ...
```

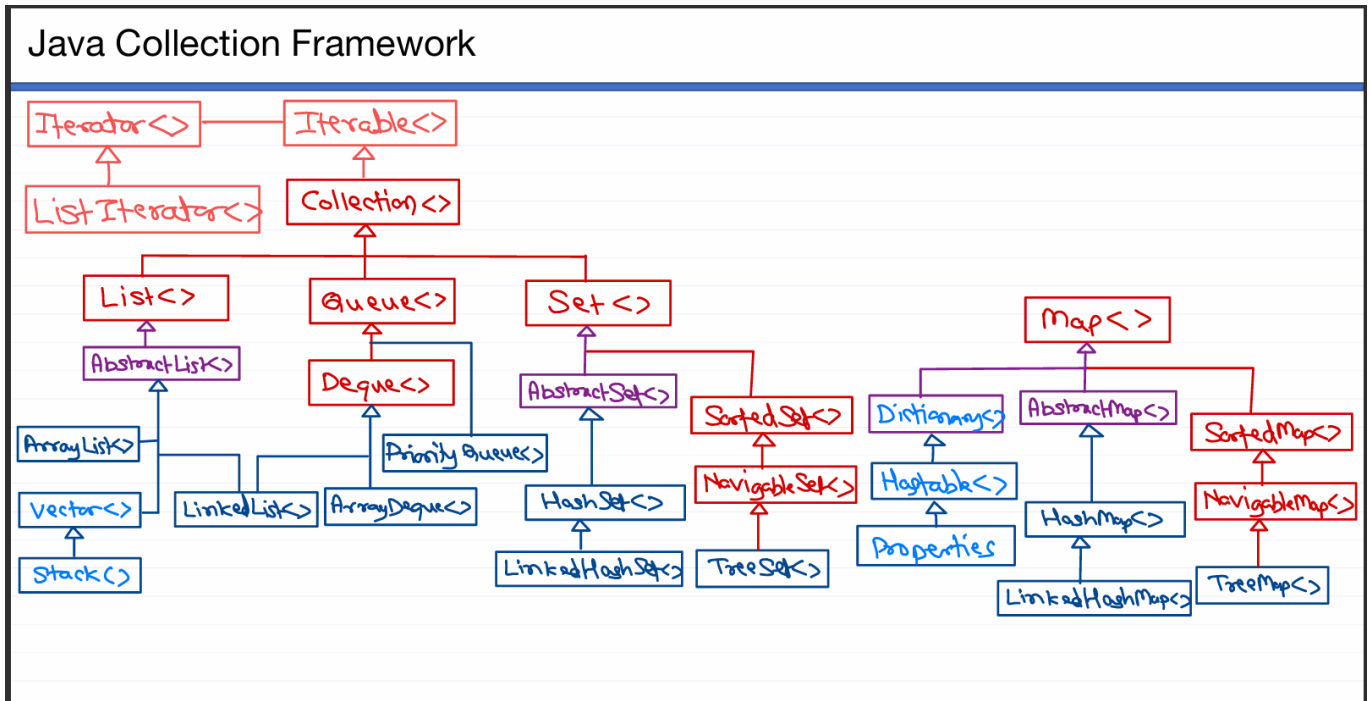
Differences:

- **Comparable standard is implemented within class by the developer of the class; while Comparator standard is implemented outside the class as per requirement.**
- Comparable is standard of comparing "this" object with "other" object; while Comparator is standard of comparing two given objects.
- java.lang.Comparable vs java.util.Comparator
- Comparable: int compareTo(T other); Comparator: int compare(T first, T second);

Java Collection Framework

- Collection framework is **Library of reusable data structure classes** that is used to develop application.

- **Main purpose** of collection framework *is to manage data/objects in RAM efficiently*.
- Collection framework was introduced in Java 1.2 and type-safe implementation is provided in 5.0 (using generics).
- **java.util** package.
- Java collection framework provides
 - **Interfaces** -- **defines standard methods** for the collections.
 - **Implementations** -- **classes that implements various data structures**.
 - **Algorithms** -- **helper methods** like searching, sorting, ...



Collection Hierarchy

- **Interfaces**: Iterable, Collection, List, Queue, Set, Map, Deque, SortedSet, SortedMap, ...
- **Implementations**: ArrayList, LinkedList, HashSet, HashMap, ...
- **Algorithms**: sort(), reverse(), max(), min(), ... -> in **"Collections" class "static" methods**.

Iterable interface

- **To traverse any collection it provides an "Iterator"**.
- **Enable use of for-each loop**.
- In **java.lang** package
- Methods
 - **Iterator iterator() // SAM (Single Abstract Method)**
 - default Splitter splitter()

- default void **forEach**(Consumer<? super T> action)

Collection interface

- *"Root interface" in collection framework interface hierarchy.*
- Most of collection classes are inherited from this interface (indirectly).
- *Provides most basic/general functionality for any collection.*
- **Abstract methods**
 - boolean add(E e)
 - int size()
 - boolean isEmpty()
 - void clear()
 - boolean contains(Object o)
 - boolean remove(Object o)
 - boolean addAll(Collection<? extends E> c)
 - boolean containsAll(Collection<?> c)
 - boolean removeAll(Collection<?> c)
 - boolean retainAll(Collection<?> c)
 - Object[] toArray()
 - Iterator iterator() -- inherited from Iterable
- **Default methods**
 - default Stream stream()
 - default Stream parallelStream()
 - default boolean removeIf(Predicate<? super E> filter)

Iterator interface

- *standard for iterating/traversing through collection.*
- **java.util**.Iterator
- Methods
 - boolean **hasNext**()
 - E **next**()
 - void **remove**() example

```
Iterator<E> e = v.iterator();
while(e.hasNext()) {
    E ele = e.next();
```

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

for-each loop vs Iterator

```
for(String ele : collection) {  
    System.out.println(ele);  
}
```

- Internally converted to following code by Java compiler.

```
itr = collection.iterator();  
while(itr.hasNext()) {  
    ele = itr.next();  
    System.out.println(ele);  
}
```

- for-each loop can be used only on the classes inherited from Iterable. Otherwise, it will give compiler error.

Iterator vs Enumeration

Enumeration

- Since Java 1.0
- Methods
 - boolean hasMoreElements()
 - E nextElement()
- Enumeration behaves similar to **fail-safe iterator**.
- Example

```
Enumeration<E> e = v.elements();  
while(e.hasMoreElements()) {  
    E ele = e.nextElement();  
    System.out.println(ele);  
}
```

ListIterator

- Part of collection framework (1.2)
- Inherited from Iterator
- **Bi-directional access**
- Methods
 - boolean hasNext()
 - E next()
 - int nextIndex()
 - **boolean hasPrevious()**
 - **E previous()**
 - **int previousIndex()**
 - void remove()
 - void set(E e)
 - void add(E e)

contains() vs equals()

```
class ArrayList ... {  
    // ...  
    public boolean contains(T key) {  
        for(T ele : values) {  
            if(key.equals(value))  
                return true;  
        }  
    }  
    return false;  
}
```

Java Collection Framework

List interface

- **Ordered/sequential collection.**
- Implementations: **ArrayList, Vector, Stack, LinkedList**, etc.
- List can contain **duplicate elements**.
- List can contain **multiple null elements**.
- Elements **can be accessed sequentially (bi-directional using Iterator) or randomly (index based)**.
- **Abstract methods**
 - void add(int index, E element)
 - String toString()
 - E get(int index)
 - E set(int index, E element)
 - int indexOf(Object o)
 - int lastIndexOf(Object o)
 - E remove(int index)
 - boolean addAll(int index, Collection<? extends E> c)
 - ListIterator listIterator()
 - ListIterator listIterator(int index)
 - List subList(int fromIndex, int toIndex)
- To store objects of user-defined types in the list, you must override equals() method for the objects. It is mandatory while searching operations like contains(), indexOf(), lastIndexOf().

ArrayList class

- **Internally ArrayList is "dynamically growable array".**
- **Elements can be traversed using Iterator, ListIterator, or using index.**
- Default initial capacity of ArrayList is **10**. If it gets filled then its capacity gets increased by **"half"** of its existing capacity.
- Primary use
 - **Random access is very fast.**

- Add/remove at the end of list.

Vector class

- **Legacy collection class** (since Java 1.0), modified for collection framework (List interface).
- Internally Vector is "**dynamically growable array**".
- Elements can be traversed using Enumeration, Iterator, ListIterator, or using index.
- Default initial capacity of vector is **10**. If it gets filled then its capacity gets increased by its **existing capacity**.
- **Synchronized collection** -- **Thread safe** but **slower performance**.
- Primary use
 - Random access (**in multi-threaded applications**)
 - Add/remove at the end of list (**in multi-threaded applications**).

NOTE:

* To perform multiple tasks concurrently within a single process, threads are used (thread based multi-tasking or multi-threading).

* When multiple threads are accessing same resource at the same time, the race condition may occur. Due to this undesirable/unexpected results will be produced.

* To avoid this, OS/JVM provides synchronization mechanism. It will provide thread-safe access to the resource (the other threads will be blocked).

Traversal

- Using Iterator
- Using for-each loop
 - Gets internally converted into Iterator traversal.
- for loop (Traversing List collection)

- Enumeration -- Traversing Vector (Java 1.0)

Synchronized vs Unsynchronized collections

- **Synchronized collections** are **thread-safe** and **sync checks cause "slower execution"**.
- Legacy collections were synchronized.
 - **Vector**
 - **Stack**
 - **Hashtable**
 - **Properties**
- Collection classes in collection framework (since 1.2) are non-synchronized (for better performance).
- **Collection classes can be converted to synchronized collection using "Collections class" methods.**
 - `syncList = Collections.synchronizedList(list)`
 - `syncSet = Collections.synchronizedSet(set)`
 - `syncMap = Collections.synchronizedMap(map)`

Collections class

- **Helper/utility class that provides several static helper methods.**
- Methods
 - List **reverse**(List list);
 - List **shuffle**(List list);
 - void **sort**(List list, Comparator cmp)
 - E **max**(Collection list, Comparator cmp);
 - E **min**(Collection list, Comparator cmp);
 - List **synchronizedList**(List list);

Collection vs Collections

Collection interface

- All methods are **public and abstract**. They implemented in sub-classes.
- Since all methods are **non-static**, must be called on object.

```
Collection<Integer> list = new ArrayList<>();  
//List<Integer> list = new ArrayList<>();  
//ArrayList<Integer> list = new ArrayList<>();  
list.remove(new Integer(12));
```

Collections class

- Helper class that contains all **static methods**.
- We never create object of "Collections" class.

```
Collections.methodName(...);
```

LinkedList class

- Internally LinkedList is doubly linked list.
- Elements can be traversed using Iterator, ListIterator, or using index.
- Primary use
 - Add/remove elements (anywhere)
 - **Less contiguous memory available**
 - Inherited from List<>, Deque<>.

Queue interface

- **Represents "utility" data structures** (like **Stack**, **Queue**, ...) data structure.
- Implementations: **LinkedList**, **ArrayDeque**, **PriorityQueue**.
- Can be accessed using iterator, **but no random access**.
- Methods
 - boolean add(E e) - throw **IllegalStateException** if full.
 - E remove() - throw **NoSuchElementException** if empty

- E element() - throw **NoSuchElementException** if empty
////////////////////////////////////
- boolean **offer**(E e) - return **false** if full.
- E **poll**() - returns **null** if empty
- E **peek**() - returns **null** if empty

- **In queue, addition and deletion is done from the different ends (rear and front).**

Deque interface

- **Represents double ended queue data structure** i.e. add/delete can be done from both the ends.
- Two sets of methods
 - **Throwing exception on failure**: addFirst(), addLast(), removeFirst(), removeLast(), getFirst(), getLast().
 - **Returning special value on failure**: offerFirst(), offerLast(), pollFirst(), pollLast(), peekFirst(), peekLast().
- **Can used as Queue as well as Stack.**
- Methods
 - boolean offerFirst(E e)
 - E pollFirst()
 - E peekFirst()
 - boolean offerLast(E e)
 - E pollLast()
 - E peekLast()

ArrayDeque class

- Internally ArrayDeque is **dynamically growable array**.
- Elements are allocated **contiguously in memory**.

LinkedList class

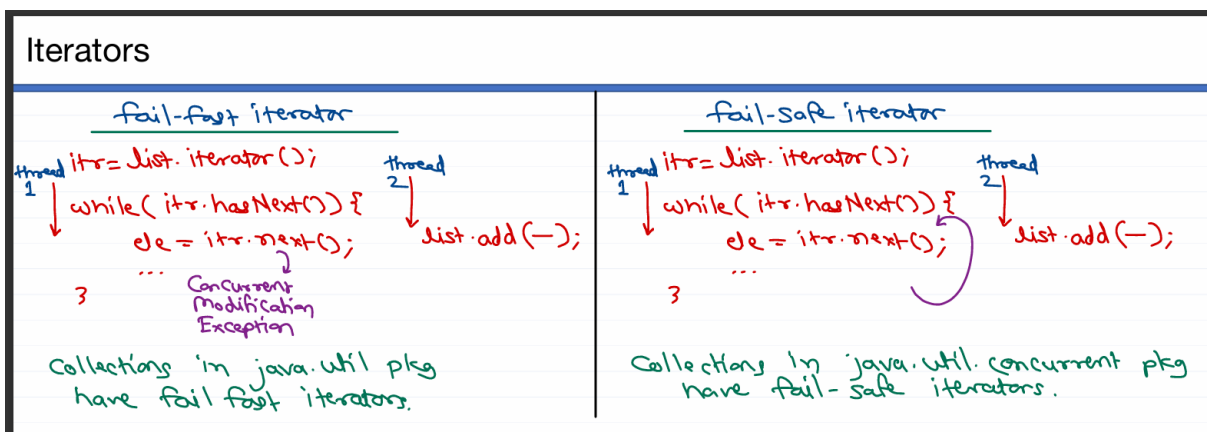
- Internally LinkedList is **doubly linked list**.

PriorityQueue class

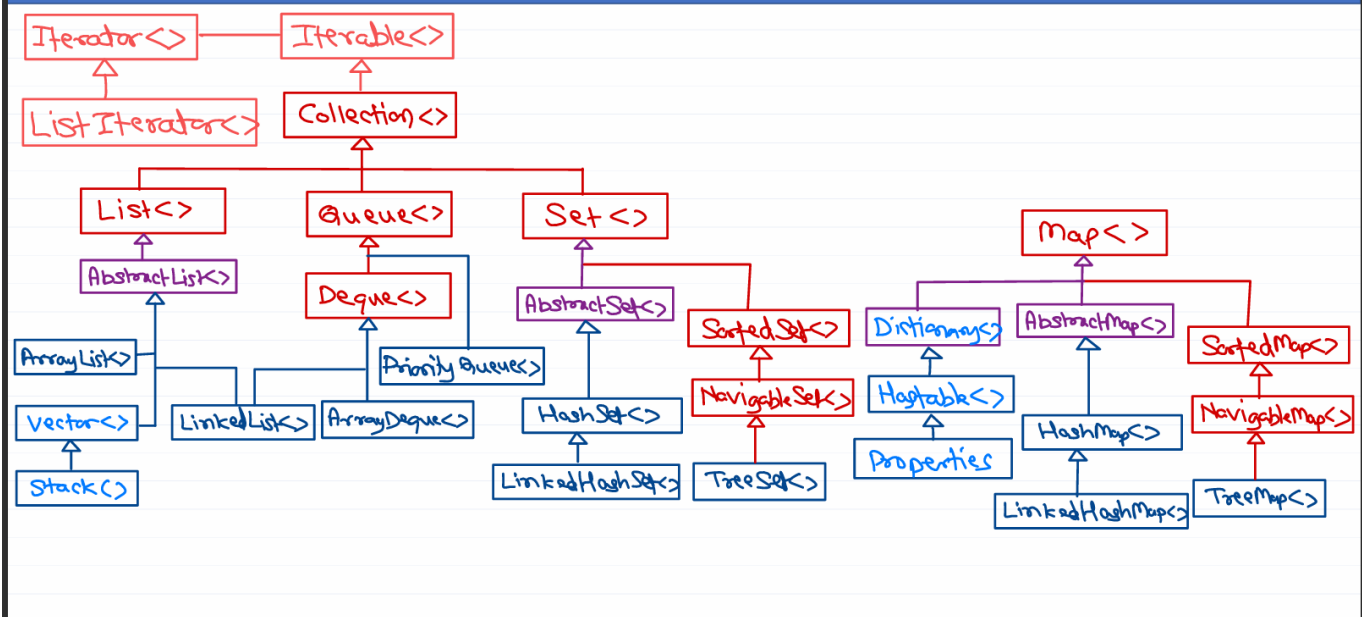
- Internally PriorityQueue is a **"binary heap"** data structure.
- Elements with highest priority is deleted first (NOT FIFO).
- Elements should have natural ordering or need to provide comparator.**

Fail-fast vs Fail-safe Iterator

- If state of collection is "modified" (add/remove operation other than iterator methods) while traversing a collection using iterator and iterator methods fails "(with ConcurrentModificationException)", then iterator is said to be Fail-fast.**
 - e.g. Iterators from ArrayList, LinkedList, Vector, ...
- If iterator "allows to modify" the underlying collection (add/remove operation other than iterator methods) while traversing a collection "(NO ConcurrentModificationException)", then iterator is said to be Fail-safe.**
 - e.g. Iterators from CopyOnWriteArrayList, ...



Java Collection Framework



Set interface

- **Collection of unique elements (NO duplicates allowed).**
- Implementations: **HashSet, LinkedHashSet, TreeSet.**
- Elements can be accessed using an Iterator.
- Abstract methods (same as Collection interface)
 - **add() returns "false" if element is "duplicate".**

HashSet class

- **Non-ordered set** (elements stored in any order).
- **Elements must implement equals() and hashCode().**
- **Fast execution.**

LinkedHashSet class

- **Ordered set (preserves order of insertion).**
- **Elements must implement equals() and hashCode().**
- **Slower than HashSet.**

SortedSet interface

- *Use natural ordering or Comparator to keep elements in sorted order.*
- Methods
 - E first()
 - E last()
 - SortedSet headSet(E toElement)
 - SortedSet subSet(E fromElement, E toElement)
 - SortedSet tailSet(E fromElement)

NavigableSet interface

- *Sorted set with additional methods for navigation.*
- Methods
 - E higher(E e)
 - E lower(E e)
 - E pollFirst()
 - E pollLast()
 - NavigableSet descendingSet()
 - Iterator descendingIterator()

TreeSet class

- *Sorted navigable set (stores elements in sorted order).*
- *Elements must implement "Comparable" or "provide Comparator".*
- *Slower than HashSet and LinkedHashSet.*

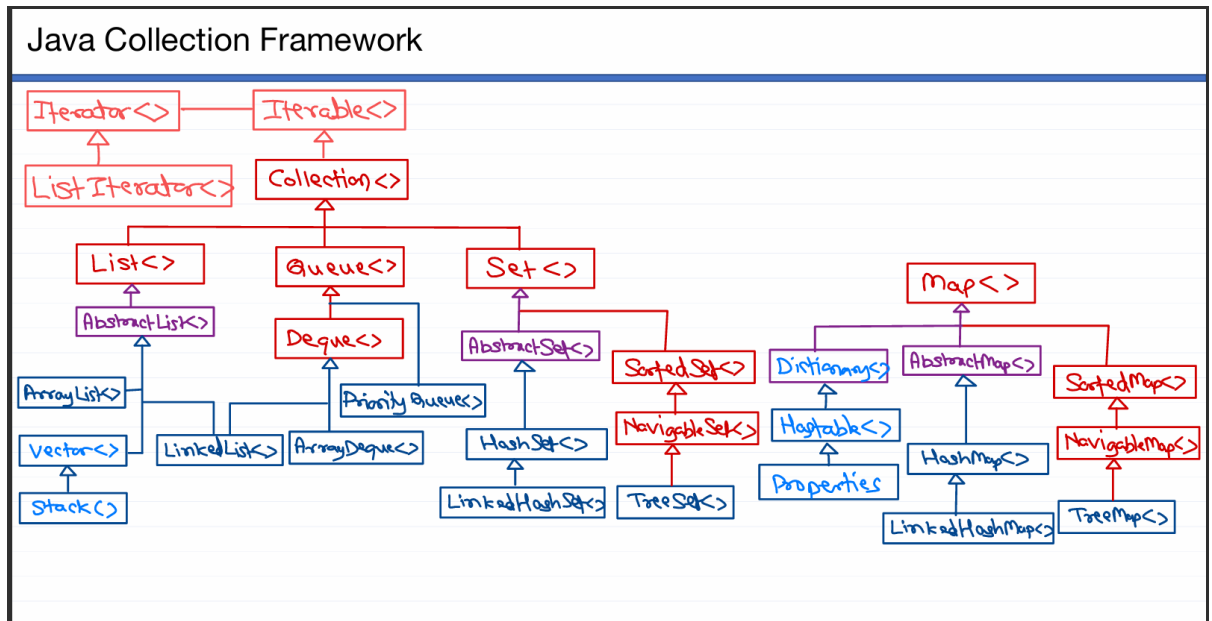
HashTable Data structure

- *Hashtable stores data in "key-value pairs" so that for the "given key", "value can be searched in fastest possible time".*
- *Internally hash-table is a "table(array)", in which "each slot(index) has a bucket(collection)". Key-value entries are stored in the buckets depending on hash code of the "key".*
- *Load factor = Number of entries / Number of buckets.*
- Examples
 - Key=Name, Value=Phone number
 - Key=pincode, Value=city/area
 - 400027 -- Byculla, Mumbai

- 411046 -- Katraj, Pune
- 411052 -- Hinjawadi, Pune
- 415110 -- Karad, Satara
- 411037 -- Marketyard, Pune
- 411002 -- Bajirao Rd, Pune
- 411007 -- Aundh, Pune
- Key=Employee, Value=Manager
- Key=Department, Value=list of Employees.

hashCode() method

- *Object class has hashCode() method, that returns a unique number for each object (by converting its address into a number).*
- *To use any hash-based data structure "hashCode()" and "equals()" method must be implemented.*
- *If two distinct objects yield same hashCode(), it is referred as "collision". More collisions reduce performance.*
- Most common technique is to **multiply field values with prime numbers** to get uniform distribution and lesser collisions.
- hashCode() overriding rules
 - hash code should be calculated on the fields that **decides equality of the object**.
 - hashCode() should return same hash code each time unless object state is modified.
 - **If two objects are equal (by equals()), then their hash code must be same.**
 - If two objects are not equal (by equals()), then their hash code **may be same** (but reduce performance).



Map interface

- **Collection of key-value entries (Duplicate "keys" not allowed).**
- Implementations: *HashMap, LinkedHashMap, TreeMap, Hashtable, ...*
- **The data can be accessed as set of keys, collection of values, and/or set of key-value entries.**
- **Map.Entry<K,V>** is nested interface of Map<K,V>.
 - K getKey()
 - V getValue()
 - V setValue(V value)
- Abstract methods
 - boolean isEmpty()
 - int size()
 - V put(K key, V value)
 - V get(Object key)
 - Set keySet()
 - Collection values()
 - Set<Map.Entry<K,V>> entrySet()
 - boolean containsValue(Object value)
 - boolean containsKey(Object key)
 - V remove(Object key)
 - void clear()
 - void putAll(Map<? extends K,? extends V> map)
- Maps not considered as true collection, because it is not inherited from Collection interface.

HashMap class

- **Non-ordered map** (entries stored in any order -- as per hash code of key).
- **Keys must implement equals() and hashCode()**.
- Fast execution.
- Mostly used Map implementation.

LinkedHashMap class

- **Ordered map (preserves order of insertion)**.
- **Keys must implement equals() and hashCode()**.
- Slower than HashSet.
- Since Java 1.4.

TreeMap class

- **Sorted navigable map (stores entries in sorted order of key)**.
- **Keys must implement Comparable or provide Comparator**.
- Slower than HashMap and LinkedHashMap
- Internally based on **Red-Black tree**.
- **Doesn't allow null key** (allows null value though).

Hashtable class

- Similar to HashMap class.
- Legacy collection class (since Java 1.0), modified for collection framework (Map interface).
- **Synchronized collection** -- **Thread safe** but **slower performance**
- Inherited from **java.util.Dictionary** abstract class (it is Obsolete).

Example

```
class Distance {  
    private int feet, inches;  
    // ...  
    public int hashCode() {  
        int hash = Objects.hash(this.feet, this.inches);  
    }  
}
```



```

        return hash;
    }
    public boolean equals(Object obj) {
        if(obj == null)
            return false;
        if(this == obj)
            return true;
        if(obj instanceof Distance) {
            if(Objects.equals(this.feet, other.feet) && Objects.equals(this.inches,
other.inches))
                return true;
        }
        return false;
    }
    /*
    public int hashCode() {
        int hash = 31 * this.feet + this.inches;
        return hash;
    }
    public boolean equals(Object obj) {
        if(obj == null)
            return false;
        if(this == obj)
            return true;
        if(obj instanceof Distance) {
            Distance other = (Distance)obj;
            if(this.feet == other.feet && this.inches == other.inches)
                return true;
        }
        return false;
    }
    */
}
class Person {
    private String name;
    private int age;
    private Distance height;
    // ...
}

```

```

class Main {
    public static void main(String[] args) {
        Map<Distance,Person> map = new HashMap<>();
        Person p1 = new Person("Nilesh", 40, new Distance(5,7));
        map.put(p1.getHeight(), p1);
        Person p2 = new Person("Rahul", 43, new Distance(5,6));
        map.put(p2.getHeight(), p2);
        // ...
        Person p = new Person();
        p.accept();
        map.put(p.getHeight(), p);
    }
}

```

```

}
}

```

Hash table in Data structure

Diagram illustrating a Hash Table structure with 10 slots (0-9). The hash function is $h(k) = k \% 10$.

Entries stored in the table:

- Slot 0: 415110 Karod Sat
- Slot 2: 411052 Hing Pun
- Slot 3: 411002 Baji Pun
- Slot 6: 411046 Kat Pun
- Slot 7: 400027 By. Mum
- Slot 8: 411037 Mark Pun
- Slot 9: 411007 Punth Pun

Collision handling techniques:

- Open addressing
- Separate chaining ✓

Load factor = num of entries ÷ num of slots

e.g.

num of entries	num of slots	load factor
7	10	0.7
10	10	1.0
15	10	1.5

Sunbeam Infotech www.sunbeaminfo.com

Java 8 Interfaces

Static methods

- Before Java 8, interfaces allowed public static final fields.
- Java 8 also allows the **static methods in interfaces**.
- They act as helper methods and thus eliminates need of helper classes like Collections, ...**

```

interface Emp {
    double getSal();
    public static double calcTotalSalary(Emp[] a) {
        double total = 0.0;
        for(int i=0; i<a.length; i++)
            total += a[i].getSal();
        return total;
    }
}

```

Default methods

- Java 8 allows default methods in interfaces. **If method is not overridden, its default implementation in interface is considered.** This allows adding new functionalities into existing interfaces without breaking old implementations e.g. Collection, Comparator, ...

```
interface Emp {
    double getSal();
    default double calcIncentives() {
        return 0.0;
    }
}
class Manager implements Emp {
    // ...
    // calcIncentives() is overridden
    double calcIncentives() {
        return getSal() * 0.2;
    }
}
class Clerk implements Emp {
    // ...
    // calcIncentives() is not overridden -- so method of interface is considered
}
```

```
new Manager().calcIncentives(); // return sal * 0.2
new Clerk().calcIncentives(); // return 0.0
```

- However default methods will lead to **ambiguity errors** as well, if same default method is available from multiple interfaces. Error: Duplicate method while declaring class.

```
interface Displayable {
    default void show() {
        System.out.println("Displayable.show() called");
    }
}
interface Printable {
    default void show() {
        System.out.println("Printable.show() called");
    }
}
class FirstClass implements Displayable, Printable { // compiler error: duplicate
    method
    // ...
}
class Main {
    public static void main(String[] args) {
        FirstClass obj = new FirstClass();
        obj.show();
    }
}
```

```
    }
}
```

- Superclass same method get higher priority. But super-interfaces same method will lead to error. Super-class wins! Super-interfaces clash!!

```
interface Displayable {
default void show() {
    System.out.println("Displayable.show() called");
}
}
interface Printable {
default void show() {
    System.out.println("Printable.show() called");
}
}
class FirstClass implements Displayable, Printable { // compiler error: duplicate
method
// ...
}
class Main {
    public static void main(String[] args) {
        FirstClass obj = new FirstClass();
        obj.show();
    }
}
```

```
interface Displayable {
default void show() {
    System.out.println("Displayable.show() called");
}
}
interface Printable {
default void show() {
    System.out.println("Printable.show() called");
}
}
class Superclass {
public void show() {
    System.out.println("Superclass.show() called");
}
}
class SecondClass extends Superclass implements Displayable, Printable {
// ...
}
class Main {
public static void main(String[] args) {
    SecondClass obj = new SecondClass();
    obj.show(); // Superclass.show() called
}
```

```
}  
}
```

- A class can invoke methods of super interfaces using `InterfaceName.super`.

```
interface Displayable {  
    default void show() {  
        System.out.println("Displayable.show() called");  
    }  
}  
  
interface Printable {  
    default void show() {  
        System.out.println("Printable.show() called");  
    }  
}  
  
class FourthClass implements Displayable, Printable {  
    @Override  
    public void show() {  
        System.out.println("FourthClass.show() called");  
        Displayable.super.show();  
        Printable.super.show();  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        FourthClass obj = new FourthClass();  
        obj.show(); // calls FourthClass method  
    }  
}
```

Functional Interface

- If interface contains exactly one abstract method (SAM), it is said to be functional interface.
- It may contain additional default & static methods. E.g. Comparator, Runnable, ...
- `@FunctionalInterface` annotation does compile time check, whether interface contains single abstract method. If not, raise compile time error.

```
@FunctionalInterface // okay  
interface Foo {  
    void foo(); // SAM  
}
```

```
@FunctionalInterface    // okay
interface FooBar1 {
    void foo();          // SAM
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface    // error
interface FooBar2 {
    void foo();          // AM
    void bar();          // AM
}
```

```
@FunctionalInterface    // error
interface FooBar3 {
    default void foo() {
        /*... */
    }
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface    // okay
interface FooBar4 {
    void foo();          // SAM
    public static void bar() {
        /*... */
    }
}
```

- Functional interfaces forms foundation for **Java lambda expressions** and **method references**.

Built-in functional interfaces

- New set of functional interfaces given in java.util.function package.
 - `Predicate<T>`: test: T -> boolean
 - `Function<T, R>`: apply: T -> R
 - `BiFunction<T, U, R>`: apply: (T, U) -> R
 - `UnaryOperator<T>`: apply: T -> T
 - `BinaryOperator<T>`: apply: (T, T) -> T
 - `Consumer<T>`: accept: T -> void
 - `Supplier<T>`: get: () -> T
- For efficiency primitive type functional interfaces are also supported e.g. `IntPredicate`, `IntConsumer`, `IntSupplier`, `IntToDoubleFunction`, `ToIntFunction`, `ToIntBiFunction`, `IntUnaryOperator`, `IntBinaryOperator`.

Anonymous Inner class

- Creates a new class inherited from the given class/interface and its object is created.
- If in static context, behaves like static member class. If in non-static context, behaves like non-static member class.
- ***Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.***

```
// (named) local class
class EmpnoComparator implements Comparator<Employee> {
public int compare(Employee e1, Employee e2) {
return e1.getEmpno() - e2.getEmpno();
}
}
Arrays.sort(arr, new EmpnoComparator());
// anonymous obj of local class
```

```
// Anonymous inner class
Comparator<Employee> cmp = new Comparator<Employee>() {
public int compare(Employee e1, Employee e2) {
return e1.getEmpno() - e2.getEmpno();
}
};
Arrays.sort(arr, cmp);
```

```
// Anonymous object of Anonymous inner class.
Arrays.sort(arr, new Comparator<Employee>() {
public int compare(Employee e1, Employee e2) {
return e1.getEmpno() - e2.getEmpno();
}
```

```
}  
});
```

Lambda expressions

- Traditionally Java uses anonymous inner classes to compact the code. For each inner class separate .class file is created.
- However, code is complex to read and un-efficient to execute. Lambda expression is short-hand way of implementing functional interface.
- Its argument types may or may not be given. The types will be inferred.
- Lambda expression can be single liner (expression not statement) or multi-liner block { ... }.

```
// Anonymous inner class  
Arrays.sort(arr, new Comparator<Emp>() {  
    public int compare(Emp e1, Emp e2) {  
        int diff = e1.getEmpno() - e2.getEmpno();  
        return diff;  
    }  
});
```

```
// Lambda expression -- multi-liner  
Arrays.sort(arr, (Emp e1, Emp e2) -> {  
    int diff = e1.getEmpno() - e2.getEmpno();  
    return diff;  
});
```

```
// Lambda expression -- multi-liner -- Argument types inferred  
Arrays.sort(arr, (e1, e2) -> {  
    int diff = e1.getEmpno() - e2.getEmpno();  
    return diff;  
});
```

```
// Lambda expression -- single-liner -- with block { ... }  
Arrays.sort(arr, (e1, e2) -> {  
    return e1.getEmpno() - e2.getEmpno();  
});
```

Prepared by: Nilesh Ghule 8 / 21

Sunbeam Institute of Information Technology, Pune and Karad day14.md

```
// Lambda expression -- single-liner  
Arrays.sort(arr, (e1,e2) -> e1.getEmpno() - e2.getEmpno());
```


Non-capturing lambda expression

- If lambda expression result entirely depends on the arguments passed to it, then it is non-capturing (self-contained).

```
BinaryOperator<Integer> op1 = (a,b) -> a + b;  
testMethod(op);
```

```
static void testMethod(BinaryOperator<Integer> op) {  
    int x=12, y=5, res;  
    res = op.apply(x, y); // res = x + y;  
    System.out.println("Result: " + res)  
}
```

- In functional programming, such functions/lambda expressions are referred to as pure functions.

Capturing lambda expression

- If lambda expression result also depends on additional variables in the context of the lambda expression passed to it, then it is capturing.

```
int c = 2; // must be effectively final  
BinaryOperator<Integer> op = (a,b) -> a + b + c;  
testMethod(op);
```

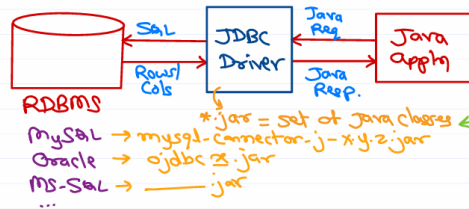
```
static void testMethod(BinaryOperator<Integer> op) {  
    int x=12, y=5, res;  
    res = op.apply(x, y); // res = x + y + c;  
    System.out.println("Result: " + res);  
}
```

- Here variable `c` is bound (captured) into lambda expression. So it can be accessed even out of scope (effectively). Internally it is associated with the method/expression.
- In some functional languages, this is known as Closures.

Java Database Connectivity (JDBC)

- **JDBC driver converts Java requests in database understandable form and database response in Java understandable form.**
- JDBC drivers are of 4 types
 - Type I - Jdbc Odbc Bridge driver
 - Type II - Partial Java/Native driver
 - Type III - Middleware/Network driver
 - Type IV - Database specific driver written completely in Java.

Java Database Connectivity



JDBC is a specification/standard. given with some 'interfaces & helper classes.'

- ① java.sql.Driver - create jdbc connection
- ② java.sql.Connection - represent jdbc connection & interact (send/rcv) with db
- ③ java.sql.Statement - represent sql stand & java.sql.PreparedStatement execute it on server. executeUpdate(), executeQuery()
- ④ java.sql.ResultSet - represent db resp = rows + cols
- ⑤ java.sql.DriverManager - choose appropriate driver for creating database connection.

JDBC Programming Steps

- ① add jdbc driver in current project/classpath.
Project → Properties → Java Build Path → Libraries → Add External Jars → Select downloaded JDBC driver jar → Apply & Close.
- ② load & register jdbc driver.
Class.forName("pkg.DriverClass"); → com.mysql.cj.jdbc.Driver
- ③ create jdbc connection.
con = DriverManager.getConnection("dburl", "dbUser", "dbPass");
- ④ create sql statement.
stmt = con.createStatement();
→ jdbc:mysql://localhost:3306/database
- ⑤ execute sql statement & process its result.
rs = stmt.executeQuery("SELECT ...");
while(rs.next()) {
 val1 = rs.getInt("col1");
 val2 = rs.getString("col2");
}
rs.close();
cnt = stmt.executeUpdate("sql");
other than SELECT
- ⑥ close stmt & connection.
stmt.close();
con.close();

SQL Injection

- Building queries by "string concatenation" is "inefficient" as well as "insecure".
- Example

```
dno = sc.nextLine();
sql = "SELECT * FROM emp WHERE deptno="+dno;
```

- If user input "10", then effective SQL will be "SELECT * FROM emp WHERE deptno=10". This will select all emps of deptno 10 from the RDBMS.
- If user input "10 OR 1", then effective SQL will be "SELECT * FROM emp WHERE deptno=10 OR 1". Here "1" represent true condition and it will select all rows from the RDBMS.
- In Java, it is recommended NOT to use "Statement" and building SQL by string concatenation. Instead use PreparedStatement.

PreparedStatement

- *PreparedStatement represents parameterized queries.*

```
String sql = "SELECT * FROM students WHERE name=?";
PreparedStatement stmt = con.prepareStatement(sql);
System.out.print("Enter name to find: ");
String name = sc.next();
stmt.setString(1, name);

ResultSet rs = stmt.executeQuery();
while(rs.next()) {
    int roll = rs.getInt("roll");
    String name = rs.getString("name");
    double marks = rs.getDouble("marks");
    System.out.printf("%d, %s, %.2f\n", roll, name, marks);
}
```

- The same PreparedStatement can be used for executing multiple queries. There is no syntax checking repeated. This improves the performance.

interface CallableStatement extends PreparedStatement: *executing stored procedures* in db -- will be discussed in next class.

- Prevent SQL injection
- More efficient execution if same query is to be executed repeatedly.